Univerza v Ljubljani
Fakulteta za računalništvo in informatiko

Izak Lipnik

Iskanje v nestrukturiranih podatkih z uporabo B-dreves nizov

diplomsko delo

univerzitetni interdisciplinarni študij
računalništva in matematike

Mentor: dr. Andrej Brodnik

Ljubljana 2014

Besedilo je oblikovano z urejevalnikom besedil LaTeX.

Izjava o avtorstvu diplomskega dela

Spodaj podpisani Izak Lipnik, z vpisno številko **63070018**, sem avtor diplomskega dela z naslovom:

Iskanje v nestrukturiranih podatkih z uporabo B-dreves nizov

S svojim podpisom zagotavljam, da:

- sem diplomsko delo izdelal samostojno pod mentorstvom dr. Andreja Brodnika,
- so elektronska oblika diplomskega dela, naslov (slov., angl.), povzetek (slov., angl.) ter ključne besede (slov., angl.) identični s tiskano obliko diplomskega dela,
- soglašam z javno objavo elektronske oblike diplomskega dela v zbirki "Dela FRI".

V Ljubljani, dne 16. junij 2014 Podpis avtorja:

Kazalo

Povzetek

V diplomski nalogi smo spoznali novo podatkovno indeksno strukturo, namenjeno iskanju v nestrukturiranih podatkih, B-drevesa znakov. Podrobneje smo se posvetili primerjavi iskanja v besedilih, med B-drevesi znakov, algoritmu priponskih dreves *ERA* ter algoritmu slovarja nizov *COSD*. Podrobneje smo se posvetili učinkovitosti uporabe pomnilniške hirearhije.

Rezultati ki smo jih dobili, so zelo zanimivi. Med vsemi tremi algoritmi, se je B-drevo znakov izkazalo kot struktura, ki omogoča najhitrejše iskanje v nestrukturiranih podatkih in hkrati najbolje izkorišča pomnilniško hirearhijo. Hkrati smo prišli do dokaza, kako pomembno je izkoriščanje pomnilniške hirearhije. Če sta se strukturi B-drevesa znakov in priponskih dreves pri merjenju rezultatov izkazali zelo podobno, tako pri merjenju časa, kot izkoriščenosti pomnilniške hirearhije, pa se je struktura slovarja nizov izkazala precej slabše pri izkoriščenosti pomnilniške hirearhije in posledično tudi pri merjenju časa.

Izkazalo se je, da je zaradi velikega števila predpomnilniških zgrešitev, do katerih je prihajalo v strukturi slovarja nizov, ta struktura porabila več časa samo za prenos podatkov v predpomnilnik, kot pa sta ostali dve strukturi porabili za celotno iskanje.

Ključne besede: B-drevesa znakov, priponska drevesa, slovarji nizov, pomnilniška hierarhija, iskanje v besedilu.

Abstract

We introduce a new text-indexing data structure, the String B-Tree, intended to search in unstructured data. We will focus on comparing string searching algorithm in String B-trees with algorithm *ERA* of Suffix Trees and algorithm *COSD* of Cache-Oblivious String Dictionarys. More specifically we focused on efficiency use of the memory hierarchies.

The results we obtained are very interesting. Between all three algorithms, the String B-tree provides the fastest search in unstructured data, and has the most efficiency use of the memory hierarchies. At the same time we've got a proof, how important is efficiency use of the memory hierarchies. If the results of the String B-tree and the Suffix Tree were very similar, the results of the String Dictionary was much worse.

It turns out that due to the large number of cache misses, which occurred in the structure of the String Dictionary, the String Dictionary spent more time just to transfer data in cache memory than the other two structures used for the entire search.

At the beginning, we'll start with problem overview of searching in strings, then we'll learn about the memory hierarchy of today's computers, to help us understand the course of algorithms for searching in strings. In the next part, we'll introduce several examples of other text-indexing data strucuters. In the main part we'll get to know the String B-Tree in detail. In short phrase, it is a text-indexing data structure, that is a combination of classic B-trees and Patricia tries. In this part we'll also get to know the algorithm of searching strings in String B-trees, and implementation of the String B-tree.

KAZALO

In the last part, we'll compare searching in strings in practice, between String B-trees, Suffix Trees and Cache-Oblivious String Dictionarys. We'll compare time and space complexity between algorithms for searching in strings.

Keywords: String B-tree, Suffix Tree, String Dictionary, memory hirearchy, searching in strings.

Poglavje 1

Uvod

1.1 Motivacija

Živimo v svetu, kjer imamo na vseh področjih ogromne količine podatkov v digitalni obliki. Če so bile velike podatkovne zbirke še pred kakšnim desetletjem velika redkost in javnosti popolnoma neznane, pa je danes popolnoma drugače. Ogromne količine elektronskih podatkov, tekstovnih zbirk, dokumentov itd. so danes dostopnejše kot kadarkoli. Danes obstajajo ogromne količine digitalnih zbirk podatkov, od elektronskih slovarjev, knjižnic, arhivov, imenikov, pa do genetskih bank ... Praktično vsaka organizacija (manjša ali večja) ima že vsaj eno podatkovno zbirko, kamor shranjuje vse podatke. Zaradi velikega razcveta digitalnih zbirk podatkov, se pojavlja potreba po učinkovitem upravljanju z njimi. Upravljanje s tovrstnimi zbirkami otežuje predvsem njihova velikost. Velikosti takšnih zbirk so včasih merili v gigabajtih, danes pa se že uporabljajo zbirke velikosti v terabajtih in ponekod celo petabajtih.

Zaradi arhitekture današnjih računalnikov je še toliko pomembneje, kako upravljamo s tako velikimi zbirkami. Današnji računalniki so sestavljeni iz več nivojev pomnilnika, ki se med seboj razlikujejo po velikosti, tehnologiji in hitrosti upravljanja s podatki. Tako danes večina računalnikov pozna minimalno tri nivoje pomnilnikov, zunanji pomnilnik, glavni pomnilnik in

predpomnilnik. Dostop do podatkov, ki so v predpomnilniku, je izredno hiter. Žal pa veliko stane prenos podatkov iz glavnega pomnilnika v predpomnilnik. Tako je ena izmed glavnih lastnosti, ki jo želimo minimizirati, število dostopov do glavnega pomnilnika in zunanjega pomnilnika. Glavni problem je torej v tem, da želimo čim manjkrat dostopati do glavnega pomnilnika, vendar pa imamo preveliko množico podatkov, da bi lahko z vsemi hkrati upravljali v predpomnilniku. Zaradi tega uvedemo posebne strukture, ki omogočajo lažje in učinkovitejše upravljanje s podatki. Primer takšne strukture je B-drevo nizov, ki omogoča hitro in učinkovito iskanje v tekstovnih podatkih. B-drevo nizov je struktura, ki uporablja kombinacijo klasičnih B-dreves in PATRICIA dreves.

1.2 Problem iskanja v zbirkah nizov

V današnjem svetu smo prišli tako daleč, da je praktično vsaka informacija zapisana v digitalni obliki. Logična posledica tega je želja po poizvedbah nad temi podatki. Osredotočili se bomo na dva različna problema iskanja v dolgih nizih, za podrobnejši opis problematike glej [2]. Za lažjo predstavitev problema bomo vpeljali naslednje oznake:

- Σ - abeceda problema
- $X[1,s]$ - niz x, dolžine s
- $X[1,i]$ - predpona niza x, dolžine i (prvih i znakov)
- $X[j,s]$ - pripona niza x (zadnjih j - s znakov)
- $X[i,j]$ - podniz niza x (znaki od vključno pozicije i do vključno pozicije j, kjer velja $1 \leq i \leq j \leq s$)

Abeceda problema Σ predstavlja množico elementov, s katerimi opišemo problem. Najbolj razširjene abecede problema so na primer vsa števila ali pa vsi znaki slovenske abecede. Moč abecede je število elementov, s katerimi

opišemo problem, na primer, moč abecede, ki jo sestavljajo vsa števila, je enaka deset.

Za iskanje v nizih se bomo osredotočili na dva problema. Za lažje razumevanje problemov, se najprej spoznajmo z določenimi osnovnimi pojmi. Predpona besede, je vsak podniz besede, ki se začne na začetku. Podniz besede, pa je vsak del besede iz katerih je sestavljena. Primer, beseda 'banana' ima predpone 'banana', 'ban', 'bana' itd, njeni podnizi pa so recimo 'banana', 'nana', 'ana' itd. Za razumevanje problema, moramo poznati še operacijo leksiografske urejenosti $\preceq_L$, ki nam omogoča urejanje besed in števil.

Problem 1 (Iskanje predpon in območna poizvedba)
Naj bo $\Delta = \{\delta_1, ..., \delta_k\}$ množica urejenih k nizov, katerih skupna dolžina je enaka N. Celotno množico shranimo na zunanji disk, kamor dostopamo do podatkov, v primeru, da jih v danem trenutku nimamo v pomnilniku. Število dostopov do zunanjega diska želimo kar se da zmanjšati, ker je v primerjavi z dostopom do glavnega pomnilnika oziroma predpomnilnika veliko počasnejše. Za dani problem moramo realizirati dve poizvedbi:

- **IsciPredpone(P)** (*PrefixSearch(P)*):

$$\text{IsciPredpone(P)} = (\delta_i, ..., \delta_j) \left\| \begin{array}{l} \text{min i: Predpona}(\delta_i) = \text{P} \\ \text{max j: Predpona}(\delta_j) = \text{P} \end{array} \right.$$

- **IsciMed(K', K")** (*RangeQuery(K', K")*):

$$\text{IsciMed}(\text{K}', \text{K}'') = (\delta_i, ..., \delta_j) \left\| \begin{array}{l} \text{min i: } \delta_i \preceq_L \text{K'} \\ \text{max j: } \delta_j \succeq_L \text{K"} \end{array} \right.$$

IscriPredpone(P) nam vrne vse nize iz množice Δ, ki imajo predpono P. Rezultat je torej podmnožica množice Δ, kjer vsi nizi vsebujejo iskano predpono P. Gre za klasičen problem, ki je pisan na kožo B-drevesu nizov, ki ga lahko rešijo v kratkem času, ne glede na dolžino teh nizov. Območna poizvedba

IsciMed(K', K") je posplošitev iskanja predpon, rešitev so vsi nizi, množice Δ, ki so leksikografsko večji ali enaki K' ter manjši ali enaki K".

Problem 2 (Iskanje vzorcev v nizih)
Naj bo zopet $\Delta = \{\delta_1, ..., \delta_k\}$ množica k nizov, katerih skupna dolžina je enaka *N*. Tokrat imamo samo eno poizvedbo:

- **Isci(P)** (SubstringSearch(P))

$$Isci(P) = (x_1, ..., x_n) \left\| \begin{array}{l} x_i \in \{\delta_1, ..., \delta_k\} \\ x_i \text{ vsebuje podniz } P \end{array} \right.$$

Dani problem je posplošitev prvega problema. Če nas je v prvem problemu zanimalo iskanje samo v začetku nizov, pa se moramo tokrat osredotočiti na celoten niz. Isci(P) nam vrne nize v vhodni množici Δ, ki vsebujejo podniz *P*. Problem je precej kompleksnejši in uporabnejši kot prvi problem, ker najde pojavitve vzorca *P* ne glede na položaj vzorca znotraj posameznega niza.

Za lažje razumevanje poizvedb si poglejmo naslednji primer. Naj bo množica Δ enaka naslednji množici:

- $\Delta := \{$*abanab*, *abbnaa*, *abnna*, *baanaba*, *nabba* $\}$

Rezultati naslednjih poizvedb bi bili tako takšni:

- IsciPredpone(*'ab'*) := {*abanab*, *abbnaa*, *abnna*}
- IsciMed(*'aba'*, *'abd'*) := {*abanab*, *abbnaa*}
- Isci(*'ab'*) := $\{$*abanab*, *abbnaa*, *abnna*, *baanaba*, *nabba* $\}$

Iz primera lažje razumemo razlike med poizvedbami. Prva poizvedba, *IsciPredpone*, je najlažja; vidimo, da je rezultat podmnožica, kjer nizi vsebujejo iskano predpono ('ab'). Druga poizvedba *IsciMed* vrne vse nize v leksikografskem redu med 'aba' in 'abd', vsi nizi v množici Δ so leksikografsko

večji ali enaki nizu 'aba', zadnja dva niza pa nista manjša ali enaka nizu 'abd', tako so rezultat prvi trije nizi množice Δ. Zadnja poizvedba pa vrne vse nize množice Δ, ker vsi nizi v množici vsebujejo podniz 'ab'.

1.3 Organizacija dela

V diplomski nalogi bomo spoznali novo podatkovno indeksno strukturo, namenjeno obdelavi besedil, B-drevesa znakov. Podrobneje se bomo posvetili primerjavi iskanja v besedilih, med B-drevesi znakov in referenčnima strukturama priponskih dreves in slovarja nizov.

V prvem delu naloge se bomo seznanili s pomnilniško hierarhijo današnjih računalnikov, da bomo lažje razumeli potek iskanja. V nadaljevanju bomo spoznali še nekaj ostalih podatkovno indeksnih struktur, ki so primerne za iskanje v besedilih.

V srednjem delu bomo podrobneje spoznali strukturo B-drevesa znakov. Gre za strukturo, ki je mešanica klasičnih B-dreves in *PATRICIA* dreves. V tem delu bomo predstavili še algoritem iskanja v B-drevesih znakov ter implementacijo B-dreves znakov.

V zadnjem delu pa bomo predstavljeno strukturo v praksi primerjali s priponskimi drevesi in slovarji nizov. Primerjali bomo časovne in prostorske zahtevnosti algoritmov pri iskanju v besedilih.

Poglavje 2

Pomnilniške hierarhije

Za razumevanje vseh problemov, ki jih srečamo pri iskanju v besedilih, moramo najprej razumeti zgradbo današnjih računalnikov. Enega izmed glavnih problemov pri iskanju v nizih najdemo v pomnilniški hierarhiji današnjih [13] računalnikov. V nalogi bomo predpostavili, da rešujemo problema v klasični pomnilniški hierarhiji [13, 6], ki jo sestavljajo trije nivoji. Model vsebuje tri vrste pomnilnika; največjemu, a hkrati najpočasnejšemu, pravimo zunanji pomnilnik in je ponavadi sestavljen iz magnetnega diska. Potem pa imamo še glavni pomnilnik [13], ki je sestavljen iz dveh delov, glavnega oziroma delovnega pomnilnika (RAM) in predpomnilnika (*Cache*). Oba sta veliko hitrejša od zunanjega pomnilnika, a hkrati tudi manjša, predpomnilnik [13] pa je hitrejši in manjši tudi od glavnega pomnilnika. Predpomnilnik vsebuje podmnožico glavnega pomnilnika, zakaj je to dobro, bomo spoznali v nadaljevanju. V večini primerov je predpomnilnik razdeljen v več nivojev, ki se razlikujejo po velikosti in hitrosti dostopov, tisti najmanjši se fizično nahajajo na istem čipu kot CPE (centralno procesna enota). Zunanji pomnilniki so danes praviloma sestavljeni iz magnetnih diskov, glavni pomnilniki iz DRAM čipov, predpomnilniki pa iz hitrejših, a dražjih, SRAM čipov. Razlog za takšno razdeljenost je preprost, magnetni diski so poceni, a počasni, nasprotno velja za SRAM čipe, DRAM čipi pa so nekje vmes med obema tehnologijama. V tabeli 2.1 lahko vidimo primerjavo hitrosti in

cene za različne pomnilniške tehnologije, kjer je cena podana v dolarjih za Gigabajt; dobimo jo tako, da ceno pomnilnika delimo z velikostjo merjeno v GB.

Pomnilniška tehnologija	Tipičen čas dostopa	Cena v $ za GB
SRAM	2-10 ns	300$ - 600$
DRAM	40-50 ns	30$ - 60$
Magnetni disk	5-15 milijonov ns	0,30$ - 0,60$

Tabela 2.1: Primerjava pomnilniških tehnologij [13]

V tabeli sicer vidimo povprečne čase dostopa za posamezne pomnilniške tehnologije, vendar nam ti dostopi v praksi ne povejo veliko. Poleg samih časov dostopa do pomnilnikov je treba upoštevati, za kakšen način dostopa gre, ali je prišlo do zgrešitve v predpomnilniku, ki ji sledi zgrešitvena kazen ...

Ena izmed glavnih lastnosti glavnih pomnilnikov je način dostopa. V praksi se uporabljajo štirje različni načini dostopa.

Naključni dostop je eden izmed osnovnih načinov dostopa. Kadar pravimo, da CPE izmenjuje podatke z glavnim pomnilnikom, ni pomembno, v kakšnem zaporedju so naslovi zapisani, saj je čas dostopa v vsakem primeru enak. Torej je glavna lastnost oziroma pravilo, da je čas dostopa vedno neodvisna spremenljivka in se ne spremeni, če spreminjamo zaporedje naslovov.

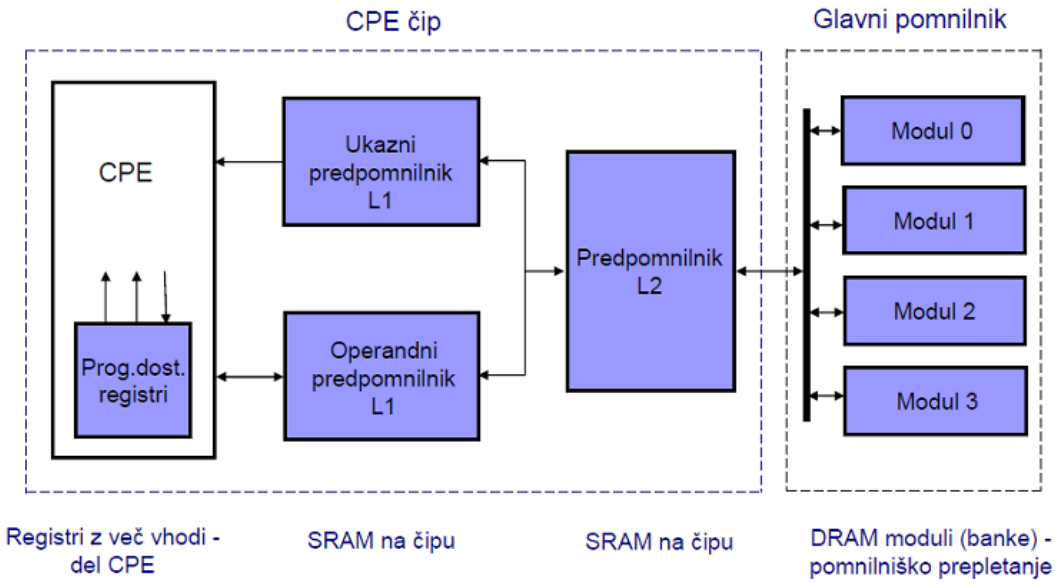
Zaporedni dostop srečamo predvsem pri magnetnih trakovnih. Čas dostopa je konstanta, kar pomeni, da ni odvisen od drugih dejavnikov. Vendar ima eno zelo veliko pomanjkljivost. Namreč, če je trak postavljen pravilno, sistemi delujejo popolnoma brez napak, če pa ni postavljen pravilno, pa se lahko zgodi, da se časi dostopa močno razlikujejo. Takšni pomnilniki so na primer magnetni trakovi, pomikalni registri in zaradi predpomninlika tudi pomninliki današnjih računalnikov.

Krožni dostop se sicer redko uporablja v računalništvu, gre za posebno vrsto zaporednega dostopa. Srečamo ga pri magnetnih diskih s fiksnimi glavami, magnetnih bobnih itd. Dostop si lahko predstavljamo kot magnetni trak, ki je zlepljen v zanko. Povprečen čas dostopa je enak polovici ene periode vrtenja.

Direktni dostop srečamo pri magnetnih diskih, ki podatke berejo s pomoč- jo premičnih glav, ki lebdijo nad diski (trdi diski). Ker so podatki zapisani v kolobarjih in razdeljeni v sektorje, je čas dostopa do njih razmeroma kratek. Previjanje pri nosilcih te kategorije ni potrebno. V resnici gre pri tem načinu za kombinacijo zaporednega in krožnega načina dostopa.

Pomnilnik bi lahko napravili in uporabljali s katerim koli načinom dostopa, vendar pa so za glavni pomnilnik najbolj primerni naključni dostopi, saj zagotavljajo najkrajši povprečni čas dostopa.

Na sliki 2.1. vidimo primer pomnilniške arhitekture z dvema nivojema predpomnilnika. Kot vidimo, se neposredno na čipu CPE nahajata oba nivoja predpomnilnika, prvi nivo predpomnilnika (L1) je razdeljen na dva dela, v ukazni in podatkovni del, drugi nivo (L2) pa je homogen. Poleg CPE čipa na sliki vidimo še čip glavnega pomnilnika, na katerem so DRAM moduli glavnega pomnilnika. Velikosti posameznih segmentov niso vnaprej določene in jih lahko uporabnik spreminja po lastni želji. Tako lahko na primer uporabnik kupi večji zunanji pomnilnik ali večji glavni pomnilnik, ne pa tudi večjega predpomnilnika. Kot smo videli, je vsaj nekaj nivojev predpomnilnika na čipu CPE, tako bi moral uporabnik z željo po večjem predpomnilniku kupiti CPE z večjim predpomnilnikom. Kot primer, štiri jedrni procesor Intel Core i7 ima štiri jedra, ki imajo vsako svoj ukazni in podatkovni pomnilnik velikosti *32*KB na prvem nivoju, prav tako ima vsako svoj homogen predpomnilnik velikosti *256*KB na drugem nivoju, na tretjem nivoju pa imajo vsa štiri jedra skupni homogeni predpomnilnik velikosti *8*MB.



Slika 2.1: Primer večnivojske pomnilniške arhitekture'

CPE seveda najhitreje operira s podatki, ki so v predpomnilniku na najvišjem nivoju, te ima praktično takoj na voljo in niso potrebna branja in prenosi iz drugih nivojev. V idealnem primeru bi program na vsaki točki izvajanja imel v predpomnilniku vse potrebne podatke. Kar pa je zaradi velikosti predpomnilnika v večini primerov nemogoče, zato želimo zmanjšati število prenosov iz zunanjega pomnilnika v glavni pomnilnik in iz tega v predpomnilnik na minimum. Hitro bi lahko prišli do zaključka, da je glavni problem pomnilniške hierarhije velikost predpomnilnika, ki je premajhen, da bi vanj lahko spravili vse podatke. Rešitev se zdi kot na dlani: povečamo predpomnilnik in CPE bo imel več podatkov dostopnih v trenutku. Vendar pa se v tej rešitvi skrivajo pasti. Če povečamo predpomnilnik, s tem upočasnimo predpomnilnik, ker je podatkov več, jih je težje najti. Osnovni problem pri gradnji pomnilnika je, da si zahtevi po velikosti in hitrosti nasprotujeta, tako je pomnilnik, če ga povečamo, počasnejši [13].

Da pa razumemo smiselnost pomnilniške arhitekture, kot smo jo opisali, moramo poznati še porazdeljenost pomnilniških naslovov, ki jih tvorijo CPE

in vhodno/izhodne naprave. Če bi bili ti naslovi porazdeljeni naključno enakomerno, predpomnilnik ne bi imel smisla, ker bi bili potrebni naslovi le redko v predpomnilniku. Vendar pa pri današnjih računalnikih ti naslovi niso naključno porazdeljeni, temveč velja zanje princip lokalnosti [13]. Princip lokalnosti pravi, da program pogosto več kot enkrat uporabi iste ukaze in operande. Posledica tega je, da se nekateri pomnilniški naslovi v določenem časovnem intervalu pojavijo veliko bolj pogosto kot drugi. V praksi se izkaže, da večina programov 90% časa uporablja samo 10% ukazov [13]. Zaradi lokalnosti lahko na osnovi znanega obnašanja programa v preteklosti precej dobro napovemo, katere naslove bo uporabljal v bližnji prihodnosti in te naslove pripravimo v predpomnilniku. V trenutku, ko CPE zahteva podatek iz pomnilnika, najprej preveri, ali iskani podatek obstaja v predpomnilniku, če ne obstaja, ga prenese iz glavnega pomnilnika. Dostopom iz glavnega pomnilnika pravimo neposredni dostopi, dostopom iz zunanjega pomnilnika pa posredni dostopi, ker potekajo preko vhodnih/izhodnih naprav. V začetku je bil osnovni razlog za delitev pomnilnika v tem, da z obstoječo tehnologijo ni bilo mogoče narediti glavnega pomnilnika, ki bi bil večji od nekaj tisoč besed. Danes so razlogi predvsem ekonomski, cena enega bita na magnetnem disku je namreč tipično 100-krat nižja kot cena v glavnem pomnilniku.

2.1 Glavni pomnilnik

Glavni pomnilniki se gradijo skoraj izključno iz elektronskih pomnilniških elementov, ki so narejeni iz majhnih silicijevih ploščic ali čipov. Vedno pa ni bilo tako, včasih so razvijalci uporabljali veliko različnih tehnologij, vendar pa se zaradi cene, hitrosti dostopa, obstojnosti in zanesljivosti danes uporabljata samo še omenjeni tehnologiji. Že od 1980-ih let dalje se glavni pomnilniki računalnikov gradijo z dinamičnimi integriranimi vezji, ki jih označujemo s kratico DRAM (dinamični RAM), poleg DRAM čipov se uporabljajo tudi statična vezja, znana pod oznako SRAM (statični RAM). SRAM čipi so hitrejši od DRAM, vendar pa so tudi manjši in dražji, zaradi česar se

manj uporabljajo. Podrobnosti o zgradbi SRAM in DRAM čipov si lahko pogledamo v [20].

Ne glede na uporabljeno tehnologijo je za računalnik najpomembnejše predvsem to, kako je glavni pomnilnik videti iz CPE in iz drugih delov računalnika. Temu pravimo organizacija pomnilnika.

Osnovna parametra vsakega pomnilnika sta pomnilniška beseda in pomnilniški naslov. Pomnilniška beseda je definirana kot najmanjše število bitov s svojim naslovom, pomnilniški naslov pa kot število, ki enolično določa neko pomnilniško besedo. Število bitov, s katerimi je podan naslov, imenujemo dolžina naslova, ki hkrati določa maksimalno velikost pomnilniškega prostora. Z izbiro dolžine besede in dolžine naslova so osnovne lastnosti pomnilnika določene. Izbira pa nikakor ni pomembna samo za pomnilnik, temveč so z njo v precejšnji meri določene tudi osnovne lastnosti računalnika v celoti. Tako je bolj ali manj nujno, da je dolžina registrov v CPE enaka dolžini pomnilniške besede ali njenemu mnogokratniku. Podobno velja tudi za širino poti med glavnim pomnilnikom na eni ter CPE in ostalimi deli računalnika na drugi strani. Večina današnjih računalnikov ima pomnilniške besede dolge 1 bajt (8 bitov), v nekaterih primerih se sicer uporablja tudi večkratnik te dolžine, npr. 32 ali 64 bitov.

Velike razlike v hitrosti in ceni pomnilniških elementov so razlog, da se pri današnjih računalnikih pomnilniki gradijo v obliki že omenjene pomnilniške hierarhije. Ne glede na to, da je delovanje računalnika s pomnilniško hierarhijo precej bolj zapleteno, so prihranki tako veliki, da se jim ni mogoče odpovedati. S pomnilniško hierarhijo lahko brez dodatnih stroškov pohitrimo delovanje računalnika, tako da zmanjšamo razliko v hitrosti med hitro CPE (centralno procesno enoto) in počasnim pomnilnikom.

2.2 Predpomnilnik

Če se za glavni pomnilnik vedno uporabljajo DRAM čipi, pa se za predpomnilnik vedno uporabijo SRAM. Ker so predpomnilniki toliko manjši

od glavnih pomnilnikov, nas višja cena glede na gigabajt ne stane veliko v primerjavi s ceno na gigabajt glavnega pomnilnika. Predpomnilnik je majhen in hiter pomnilnik, ki ga priključimo med CPE in glavni pomnilnik, lahko pa ga imamo tudi drugod, npr. pri V/I procesorjih ali pri V/I napravah. Angleška beseda *cache* pomeni varen prostor za skrivanje ali hranjenje stvari. Predpomnilnik vsebuje podmnožico naslovov, ki so v glavnem pomnilniku. Predpomnilnik je veliko manjši od glavnega pomnilnika, tipično je manjši kot 1% velikosti glavnega pomnilnika. Zakaj je smiselna tako majhna velikost predpomnilnika, smo spoznali že v začetku poglavja, in sicer zaradi principa lokalnosti, računalnik večji del izvajanja določenih programov uporablja iste pomnilniške lokacije, tako lahko le-te shrani v predpomnilnik in do njih dostopa zelo hitro. Predpomnilnik je danes razdeljen v več nivojev, na prvem nivoju pa je včasih še predpomnilnik razdeljen v dva dela, v ukazni in podatkovni predpomnilnik. Tej delitvi pravimo Hardvardska arhitektura, predpomnilniku pa nehomogen predpomnilnik. Če predpomnilnik ni deljen, mu pravimo homogen.

Predpomnilnik je sestavljen iz dveh delov, iz kontrolnega in pomnilniškega dela. Pomnilniški del je razdeljen v enote enakih velikosti, ki jim pravimo bloki. Blok ni nič drugega kot 2^b sosednjih pomnilniških besed. Velikost bloka B je lahko tudi 1, vendar pa so v večini predpomnilnikov večji, tipično od *8* do *256* pomnilniških besed. Kontrolni del predpomnilnika vsebuje informacijo, ki enolično opisuje vsak blok, ta informacija mora vsebovati vsaj naslov bloka, običajno pa so v njej še dodatni kontrolni biti.
Uspešnost predpomnilnika merimo z verjetnostjo zgrešitve. Do zgrešitve pride vedno, ko v predpomnilniku iščemo podatek, ki ga v danem trenutku ni v predpomnilniku. Takrat je potrebna zgrešitvena kazen, ki vsebuje prenos bloka iz glavnega pomnilnika v predpomnilnik. Najprej je potrebno najti blok v glavnem pomnilniku, ki ga iščemo, ter ga zamenjati z enim izmed blokov v predpomnilniku. Poznamo dve strategiji, s katerima izberemo, kateri blok v predpomnilniku bomo zamenjali, prva je naključna strategija, druga pa strategija LRU (*least recently used*). Zaradi preprostosti je precej bolj

priljubljena prva strategija, ki naključno izbere, kateri blok bomo zamenjali, medtem ko druga strategija vedno izbere tisti blok, ki je bil nazadnje uporabljen.

2.3 Časi dostopov

2.3.1 Glavni pomnilnik in predpomnilnik

CPE sproži iskanje pomnilniške besede, najprej preveri, ali je beseda shranjena v predpomnilniku, v kolikor je ni, jo išče v glavnem pomnilniku. Če iskane besede ni v predpomnilniku, temu pravimo zgrešitev, takrat je podatke potrebno iskati na višjih, počasnejših nivojih. Uspešnost delovanja predpomnilnika merimo z verjetnostjo zgrešitve (*miss rate*), *1 - H*, oziroma verjetnostjo zadetka H (hit ratio). Če je pri N_p od skupno N dostopov informacija v predpomnilniku, je verjetnost zadetka enaka:

- $H = \frac{N_p}{N} = \frac{N_p}{N_p+N_g}$.

Pri preostalih $N_g = N$ - N_p dostopih imamo zgrešitev in potreben je dostop do glavnega pomnilnika. V praksi danes, v povprečju dosegamo predpomnilniške verjetnosti zadetka večje od 0,9, običajno celo od 0,95. Če označimo s t_g čas dostopa do glavnega pomnilnika in s t_p čas dostopa do predpomnilnika, lahko izrazimo povprečen čas dostopa, ki velja za pomnilnik in predpomnilnik skupaj z naslednjo formulo:

- $t_a = t_p + (1 - H)t_g$.

Čas za dostop do predpomnilnika t_p je potreben vedno (tudi pri zgrešitvi), ker je dostop potreben za ugotavljanje zadetka ali zgrešitve. Ker je t_p veliko manjši kot t_g, hitro ugotovimo, da predpomnilnik precej izboljša povprečni čas dostopa do pomnilnika. Moramo pa povedati, da takšen izračun ni preveč uporaben, ker nismo upoštevali zgrešitvene kazni. Zgrešitvena kazen je čas, ki se pri zgrešitvi prišteje času dostopa do glavnega pomnilnika, takrat namreč ni dovolj, da samo najdemo podatek v glavnem pomnilniku, ampak ga je

potrebno še prenesti v predpomnilnik.
Zgrešitvena kazen je odvisna od zgradbe predpomnilnika, širine podatkovnih poti med predpomnilnikom in glavnim pomnilnikom ter od morebitnega drugega nivoja predpomnilnika. Glavna lastnost pri zgrešitveni kazni pa je velikost bloka B, ta namreč določi, koliko zaporednih pomnilniških besed se bo ob zgrešitvi zamenjalo. Med glavnim pomnilnikom in predpomnilnikom se namreč vedno zamenja celoten blok. Danes so velikosti bloka B tipično od 4 pa do 512 pomnilniških besed.
Predpomnilnik pozitivno vpliva tudi na hitrost CPE, na računalnikih s predpomnilnikom se povprečni čas dostopa do pomnilnika zmanjša, posledično se poveča hitrost CPE. Tu je sicer potrebno povedati, da je tudi v primeru zadetka v predpomnilniku dostop do registrov na CPE še vedno hitrejši kot do predpomnilnika (sicer registri ne bi bili potrebni). Branje iz registrov in predpomnilnika je sicer primerljivo, vendar pa lahko dostopamo do več registrov hkrati, kar pri predpomnilniku ni možno. Poznamo več načinov za zmanjšanje kazni ob predpomnilniški zgrešitvi, kot so ,,vnaprejšnji prevzem bloka" in ,,neblokirajoči predpomnilnik", vendar pa jih v celoti ne znamo odpraviti.

2.3.2 Zunanji pomnilnik

Za razliko od dostopa do glavnega pomnilnika in predpomnilnika imamo pri dostopu do zunanjega pomnilnika, ki ga ponavadi predstavlja trdi disk, v večini primerov direktni način dostopa. Dostop do podatkov na disku je sestavljen iz naslednjih treh korakov:

1. **Iskanje**. Pomik glave na želeno sled imenujemo iskanje, čas za pomik pa iskalni čas. Povprečni iskalni čas je enak vsoti časov za vsa iskanja, deljeni s številom vseh možnih iskanj. Povprečni iskalni časi so tipično med *3* in *10* ms.

2. **Vrtilna zakasnitev**. Ko je glava na pravi sledi, je potrebno počakati, da se želeni sektor zavrti pod glavo. Temu času pravimo vrtilna

zakasnitev (*rotational latency*). Povprečna vrtilna zakasnitev je enaka polovici časa, potrebnega za en obrat diska. Danes se večina diskov vrti s *7200* obrati na minuto, na takšnih diskih znaša vrtilna zakasnitev 4,2 ms.

3. **Prenos podatkov**. Čas za prenos bitov, ki sestavljajo podatkovni zapis sektorja, imenujemo notranji prenosni čas (*transfer time*). Ta čas je odvisen od hitrosti vrtenja, premera plošč in od gostote zapisa. Tipične notranje hitrosti prenosa so med *500* in *2000* Mbit/s. Ker imamo ponavadi vgrajen predpomnilnik, v katerem so shranjeni sektorji, ki so se nazadnje uporabljali, je zunanja hitrost prenašanja podatkov med pomnilnikom in diskom običajno večja od 100 MB/s.

Sedaj poznamo vse potrebne podatke, da lahko za primer izračunamo povprečen čas za branje ali pisanje *512*-bajtnega sektorja pri disku, ki se vrsti s *7200* obr/min. Povprečni iskalni čas je *8* ms, notranja prenosna hitrost pa 1000 Mbit/s. Če predpostavimo, da disk ni zaseden, je povprečni čas t_a enak:

- $t_a = 8 \text{ ms} + \frac{0{,}5*60*1000}{7200} \text{ ms} + \frac{512*8}{1000000} \text{ ms} = 12{,}17 \text{ ms}.$

Vidimo, da od 12,17 ms porabimo samo 0,004 ms za prenos bitov sektorja, preostali čas je samo za čakanje pomika in vrtenja.

Poglavje 3

Strukture za iskanje v nizih

Zaradi razširjenosti problema iskanja v nizih danes obstaja veliko različnih rešitev. Strukturam, ki so namenjene iskanju v velikih nizih, pravimo podatkovno indeksne strukture (*indexing data structures*). V splošnem je tako, da ima vsaka rešitev svoje slabosti in svoje prednosti, nekatere so hitrejše pri iskanju v nizih in počasnejše pri vstavljanju ali brisanju nizov iz strukture, spet druge dobro delujejo v praksi, a so počasne v najslabših primerih itd. Strukture lahko razdelimo v dve skupini, glede na to, ali so namenjena delovanju v zunanjem pomnilniku (trdi disk) ali v glavnem (RAM). Vse strukture sicer lahko delujejo tako v zunanjem kot notranjem pomnilniku, a za ceno slabših rezultatov. Sedaj si bomo ogledali lastnosti nekaterih osnovnih podatkovno indeksnih struktur.

Obratne datoteke (*inverted files*) so pomembna tehnika indeksiranja. Posebnost tehnike je v tem, da se vloga zapisov in atributov zamenjata. Tako potrebujemo za pridobivanje zapisa njegove atribute in ne obratno. Podrobnejši opis strukture obratnih datotek opisuje članek [21]. Obrnjeno vlogo ključa in zapisa imenujemo obratni seznam [10, 12], njegova glavna prednost je v tem, da zavzame izredno malo prostora. Problem 1 in problem 2 (omenjena v poglavju 1) lahko rešimo z obratnimi datotekami tako, da kot zapis vzamemo tekste, kot ključe pa podnize teksta. Žal se izkaže, da v praksi

tehnika ne deluje najbolje; predvsem je težko pridobiti ključe pri obravnavi nestrukturiranih besedil (npr. DNK sekvence). Poleg tega pa je težko paziti, da atributi ne vsebujejo preveč podvojenih informacij. Zaradi teh lastnosti se izkaže, da je zahtevnost reševanja problema 2 s to tehniko precej visoka, ker v najslabšem scenariju zahteva zelo veliko dostopov do zunanjega pomnilnika. V praksi se tehnika obratnih datotek izvede v zgoščenih tabelah ali različnih drevesih.

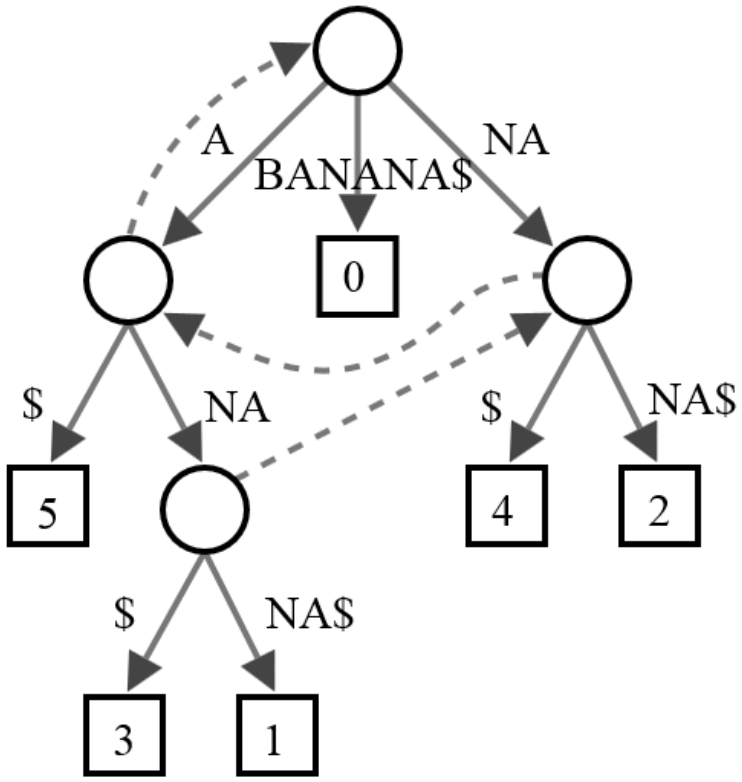
Predponska B-drevesa (prefix b-trees) so poseben tip B-dreves [4]. Listi so enaki ključem, notranji elementi pa vsebujejo kopije nekaterih ključev, ki so potrebne za potovanje od korena pa do lista drevesa. Največji problem teh dreves je v velikosti ključev. Če so ključi preveliki, namesto njih uporabimo logične kazalce, ki so precej manjši, hkrati pa lahko z njihovo pomočjo pridemo do ključa. Preslikavo ključev ponavadi uporabimo takrat, ko so ključi večji od B in jih ne moremo shraniti na eno stran pomnilnika. Prav tako lahko v notranja vozlišča namesto ključev shranjujemo samo delilnike, to so katerikoli nizi, ki so v leksikografskem redu med obema elementoma. Primer, za niza 'drevo' in 'polje' je lahko delilnik niz 'drevo', niz 'f' ali katerikoli niz, ki v leksikografskem redu leži med njima. Prednost delilnikov je v zmanjšani prostorski zahtevnosti, zato se na vsakem koraku odločimo za najkrajši delilnik, da prihranimo kar se da veliko prostora. Za izbiro delilnikov poznamo dve strategiji, prva uporabi najkrajšo unikatno predpono niza kot njegov delilnik, vendar pa se v praksi za ta pristop izkaže, da vsebuje veliko podvojenih informacij, zaradi tega, ker imajo sosedi najdaljše skupne predpone. Druga strategija uporablja stisljivo shemo za shranjevanje ključev v notranja vozlišča, kot Unix-ova B-drevesa [24]. Ta, če je prvih n znakov enakih prvim n znakom predhodnika, namesto teh n znakov zapiše številsko vrednost n, na koncu pa doda še znake od pozicije n+1 pa do konca niza. Pristop sicer zmanjša prostorsko zahtevnost, kljub temu pa ne zagotavlja, da ne bodo nizi imeli velikega števila znakov od pozicije n naprej. V najslabšem primeru je preponsko B-drevo [5, 3] učinkovito takrat,

ko imamo opravka z omejenimi velikostmi nizov (npr. do 255 znakov). Učinkovitost pa se precej poslabša v najslabšem primeru, ko imamo opravke z neomejenimi velikostmi nizov.

Priponska polja (*suffix arrays*) omogočajo hitra iskanja, ki so neodvisna od velikosti abecede problema. Priponska polja [9, 15] vsebujejo v urejenem leksikografskem redu logične kazalce na vse podnize nizov. Logični kazalec je naslov v pomnilniku, kjer se podniz začne. Naj bo niz '*banana*' niz, nad katerim bomo zgradili priponsko polje. V polje moramo tako vstaviti šest podnizov (niz je dolžine pet, šesti znak pa predstavlja poseben znak $, za konec niza), in sicer podnize {*'banana$', 'anana$', 'nana$', 'ana$', 'na$', 'a$', '$'*}. V tem primeru bi bilo priponsko polje enako polju [*6, 5, 3, 1, 0, 4, 2*], ki bi predstavljalo podnize urejene po naraščajočem leksikografskem redu [*'$' 'a$', 'ana$', 'anana$', 'banana$', 'na$', 'nana$'*]. Iz prostorskega pogleda so priponska polja najbolj učinkovita podatkovno indeksna struktura, ker zavzame najmanj prostora in nima omejitev glede dolžin ključev. Kljub temu pa se pojavi problem, če so nizi zelo veliki, velikost polj lahko naraste, tako da jih ne moremo uporabljati v predpomnilniku. Hkrati so polja, ko imamo tako dolge nize zelo težka za posodabljanje, kar pomeni, da je včasih lažje zgraditi novo polje, kot posodobiti starega. Pred kratkim so v članku [7] predstavili dinamične verzije predponskih polj, ki lahko delujejo tudi v predpomnilniku. Te lahko sicer razširimo, da delujejo v zunanjem pomnilniku, a s tem izgubijo prostorsko optimalnost in dosežejo slabše čase pri iskanju.

Priponska drevesa (*suffix trees*) in stisljiva številska drevesa (*compressed trie*) [19] so elegantna in močna podatkovna indeksna struktura, ki se pogosto uporabi za problem iskanja v nizih. Priponsko drevo [11, 14, 1] je stisljivo številsko drevo(*compressed trie*), zgrajeno nad vsemi podnizi besedila. Vsaka povezava drevesa je označena z skupno predpono vseh nizov v otrocih. Primer priponskega drevesa vidimo na sliki 3.1, ki je zgrajen za niz *'banana$'*, vsi podnizi so torej kot v primeru priponskih polj enaki {*'banana$', 'anana$',*

'nana$', 'ana$', 'na$', 'a$', '$'}.



Slika 3.1: Primer priponskega drevesa za niz 'BANANA$'

Drevo začnemo graditi v korenu, korenu nato dodamo toliko naslednikov, kot je različnih predpon vseh nizov. Kot vidimo, imamo tri različne predpone, prve se začnejo s črko *'a'*, druge se začnejo s črko *'b'*, in ker je takšna samo ena, lahko dodamo kar celoten niz, *'banana$'*, tretje pa se začnejo s črko *'n'*; in ker se s predpono *'n'* začneta samo niza *'nana$'* in *'na$'*, lahko vstavimo skupno predpono *'na'*.

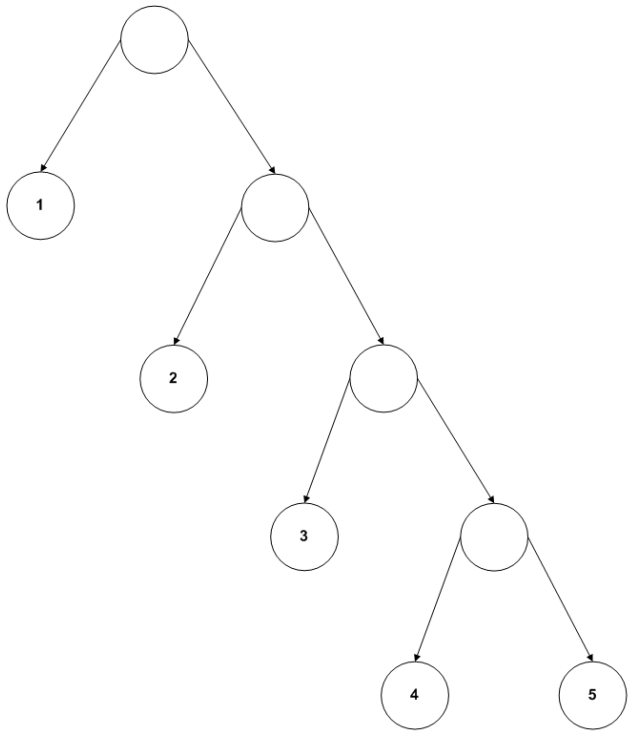
Postopek ponovimo v vsakem nasledniku, dokler ne pridemo tako daleč, da nam je ostal sam en niz. Ko nam ostane še samo en niz, vozlišče označimo kot list (na sliki kvadrat) in vanj zapišemo logični kazalec. Nobeno vozlišče nima samo enega naslednika, vsak niz pa je dobljen tako, da ga sestavljamo (dodajamo na konec) po poti od korena do lista. S tem, ko na konec vsakega

niza dodamo znak za konec, zagotovimo relacijo ena proti ena, da ima vsak niz natanko en list in obratno.

V drevesih imamo še eno posebnost, ki ji pravimo priponska povezava (*suffix link*) [16]. Gre za kazalce iz enega vozlišča v drugega, ki so v pomoč pri operacijah iskanja in posodabljanja v drevesu. Naj bo S enak nizu trenutnega vozlišča, ki ga dobimo tako, da se iz korena sprehodimo do vozlišča in pri tem nizu dodajamo oznake povezav na konec. Niz S_{-1} pa naj bo enak nizu S brez prvega znaka. Priponske povezave napravimo od vozlišč, do katerih vodi niz S, pa do vozlišč, do katerih vodi niz S_{-1}. Na sliki lahko vidimo primer treh priponskih povezav, prva vodi iz levega naslednika korena, ki predstavlja niz *'a'*, pa do korena, ki predstavlja prazen niz. Druga priponska povezava vodi od tretjega naslednika korena, ki predstavlja niz $S =$ *'na'*, pa do prvega naslednika korena, ki predstavlja niz $S_{-1} =$ *'a'*. Zadnja priponska povezava vodi med vozliščem, ki predstavlja $S =$ *'ana'*, v vozlišče, ki predstavlja S_{-1} $=$ *'na'*.

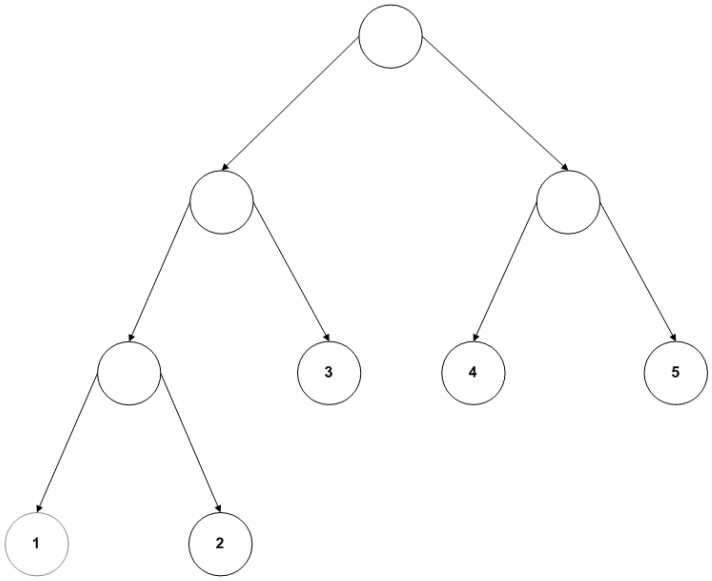
Podobno kot ostale strukture imajo priponska drevesa težave, ko je velikost besedila, v katerem iščejo, tako velika, da je ne moremo spraviti v glavni pomnilnik. Glavni razlogi za to težavo so:

a) Drevesa lahko imajo zelo neuravnoteženo topologijo, ker so odvisna od zgradbe besedila. Tako je lahko v najslabšem primeru topologija zelo neuravnotežena, kar pomeni, da je iskanje določenih nizov (najpogostejših nizov) veliko zahtevnejše, kot iskanje ostalih. Drevo je uravnoteženo takrat, ko so listi shranjeni na maksimalno dveh nivojih, neuravnoteženo pa sicer. Primer neuravnoteženega drevesa vidimo na sliki 3.2.



Slika 3.2: Primer neuravnoteženega drevesa

Na tej sliki se lepo vidi, zakaj je iskanje določenih nizov zahtevnejše. Če bi iskali niz, ki je shranjen v listu na prvem nivoju (prvi naslednik korena), bi potrebovali samo eno iskanje v vozlišču. Če pa bi iskali niz, ki je shranjen v listu na zadnjem nivoju, bi potrebovali kar štiri iskanja v vozlišču, oziroma če bi bil niz shranjen v listu na nivoju H, bi potrebovali kar H več iskanj. Na naslednji sliki 3.3 pa vidimo primer uravnoteženega drevesa za isto strukturo besedila.



Slika 3.3: Primer uravnoteženega drevesa

Na tej sliki je lepo vidno, da je maksimalna višina drevesa enaka tri, tako potrebujemo za katerokoli iskanje maksimalno tri iskanja znotraj vozlišč.

Če z H označimo višino drevesa, imamo lahko v neuravnoteženem drevesu nize, za katere potrebujemo do H iskanj v vozliščih, medtem ko v uravnoteženem drevesu potrebujemo maksimalno $log\ H$ iskanj v vozliščih.

b) Ker je stopnja notranjih vozlišč lahko poljubno velika, se lahko zgodi, da vseh kazalcev vozlišča ne spravimo na eno stran v zunanjem pomnilniku (trdem disku), kar pomeni, da mora biti vozlišče shranjeno na več strani v pomnilniku. Posledično to pomeni več dostopov do zunanjega pomnilnika, ki upočasnjujejo iskanje. Primer, stran na magnetnem disku, oziroma zunanjem pomnilniku, je ponavadi velikosti 4Kb. Zaradi tega si želimo biti sposobni vsako vozlišče shraniti na velikost 4Kb, da potrebujemo branje samo ene strani, ko želimo vozlišče

prebrati, in s tem zmanjšati število dostopov na zunanji pomnilnik. Ker pa stopnja notranjih vozlišč priponskih dreves ni omejena, se lahko zgodi, da ima vozlišče toliko naslednikov, da nismo sposobni vozlišča zapisati na eno stran zunanjega pomnilnika, kar pomeni, da moramo ob branju vozlišča večkrat dostopati do zunanjega pomnilnika. V praksi se to zgodi takrat, ko imamo veliko nizov ali podnizov z isto predpono.

c) Potovanje od vozlišča do enega izmed njegovih naslednikov zahteva dodatni dostop do diska oziroma zunanjega pomnilnika. Ker so vozlišča shranjena na zunanjem disku, so nasledniki predstavljeni s kazalci na zunanji disk. Za vsako pridobitev vozlišča, ki ga označuje povezava, je potrebno dostopati do diska (če je vozlišče preveliko za eno stran na disku, potrebujemo celo dva dostopa).

Zaključek je podoben kot v prejšnjih strukturah; struktura je sicer primerna za večji del primerov iskanja v velikih nizih, vendar pa reševanje problema 1 in 2 ni učinkovito v vseh primerih in vseh porazdelitvah besedil (neuravnoteženost, prevelika vozlišča ...). Priponska drevesa so sicer skoraj identična struktura priponskim poljem, iskanje v priponskem polju lahko implementiramo na priponskih drevesih in obratno. Priponska polja imajo nekaj prednosti v primerjavi s priponskimi drevesi, predvsem prostorsko optimalnost, vendar pa so na drugi strani priponska drevesa lažja za implementacijo in lažje razumljiva.

Poglavje 4

Struktura B-drevesa nizov

Kot smo videli, imajo vse podatkovno indeksne strukture, predstavljene v prejšnjem poglavju, slabost, da se ne obnesejo najbolje v najslabših primerih (worst case scenario). Sedaj si bomo ogledali novo strukturo B-drevesa nizov [8], ki je prav tako podatkovno indeksna struktura, vendar pa se za razliko od ostalih dobro odreže tudi v najslabših primerih.

Za lažje nadaljevanje bomo postavili določene predpostavke. Predpostavljamo, da je množica k nizov Δ shranjena na zunanjem pomnilniku na zaporedne strani trdega diska. Za predstavitve nizov bomo uporabili logične kazalce, ki kažejo na mesto v disku, kjer je shranjen posamezni niz oziroma podniz. Ko upravljamo z logičnimi kazalci dolgih nizov, lahko sicer na eno stran diska (velikosti B), shranimo Θ(B) logičnih kazalcev na nize, vendar pa je potrebno za pridobitev posameznega niza ponovno dostopati do zunanjega pomnilnika. Hitro vidimo, da bomo potrebovali veliko število dostopov do zunanjega pomnilnika, ki so počasni in bodo predstavljali ozko grlo iskanja v B-drevesih nizov. Naša glavna naloga bo zmanjšati število dostopov do zunanjega pomnilnika. Dva niza lahko primerjamo klasično, znak za znak, vendar je to izredno neučinkovito, ker je potrebno vsakič znova pridobivati nize iz pomnilnika. Potrebi po večkratnem dostopu do istega niza na zunanji pomnilnik pravimo "odvečni dostopi do pomnilnika"(rescanning), ker je potrebno večkrat pridobiti isti zapis. Verjamemo, da je ena izmed ključnih

lastnosti za učinkovito reševanje problema 1 in 2 predstavljenih v uvodu, minimalno število ponovnih pridobivanj oziroma odvečnih dostopov do pomnilnika.

4.1 B - drevo

Za začetek si poglejmo poenostavljano strukturo B-dreves nizov, ki jo bomo potem postopoma nadgrajevali v končno strukturo B-dreves nizov.
Prva pomembna lastnost:

- parameter B je vedno izbran tako, da lahko vsako vozlišče drevesa shranimo na eno stran zunanjega pomnilnika (trdega diska).

S to lastnostjo se izognemo nepotrebnemu večkratnemu branju iz zunanjega pomnilnika, ko bi bilo potrebno za eno vozlišče iz zunanjega pomnilnika prebrati več strani. Ko izberemo velikost parametra B, ki je odvisna od velikosti B zunanjega pomnilnika, lahko uvedemo naslednjo lastnost:

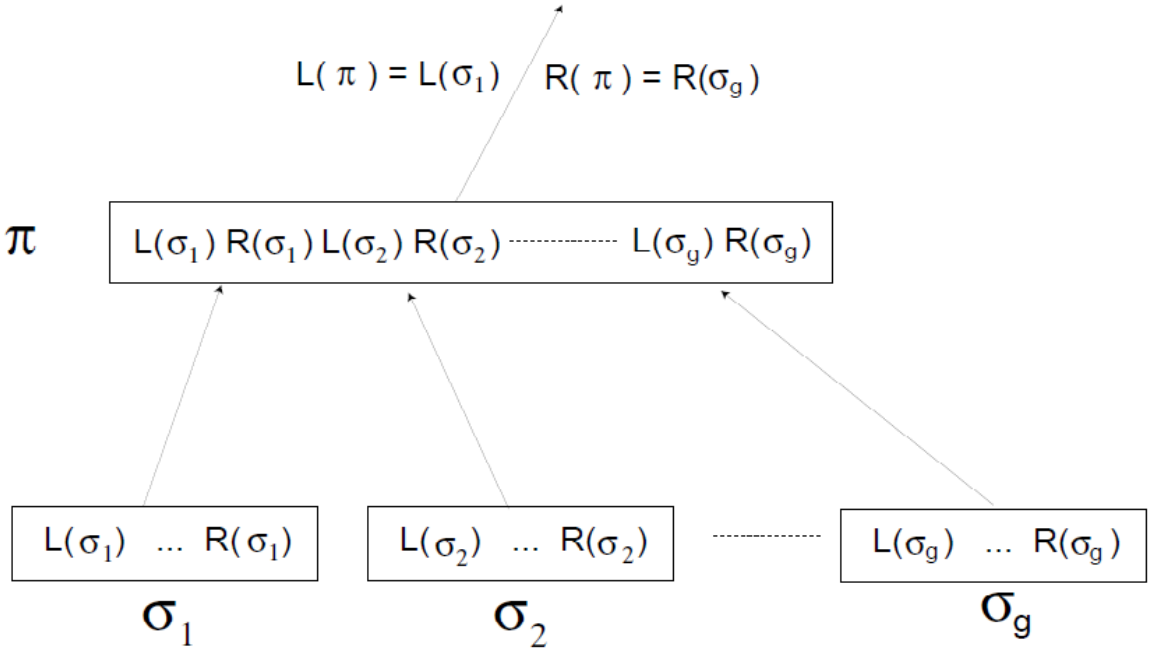
- $B \leq |S_\pi| \leq 2B$,

kjer S_π predstavlja množico nizov, ki so shranjeni v vozlišču π. Torej vsako vozlišče ima lahko shranjeno od B pa do $2B$ elementov oziroma nizov. Izjema je koren, ki mu dovoljujemo, da ima manj kot B nizov. Ko pravimo, da vozlišče vsebuje nize, mislimo na logične kazalce nizov, ker ne shranjujemo vedno celotnih kopij nizov, ker bi zavzelo preveč pomnilniškega prostora.

Strukturo bomo najlažje razumeli, če začnemo v najnižjih delih drevesa, torej listih. Predpostavimo, da je naša množica nizov Δ leksikografsko urejena. Najprej razdelimo Δ v skupine s po B nizi oziroma logičnimi kazalci nanje, z izjemo zadnje skupine, ki ima lahko med B in $2B$ nizi. Vsako skupino shranimo v nov list, v takšnem vrstnem redu, da se ohrani leksikografska urejenost.
Na tej točki uvedemo naslednjo lastnost:

- notranja vozlišča imajo od $\frac{B}{2}$ pa do B otrok

Sedaj vemo, koliko elementov in koliko otrok lahko imajo vozlišča. Po tem, ko smo spoznali zgradbo listov, si poglejmo še zgradbo notranjih elementov. V vsako vozlišče združimo od $\frac{B}{2}$ pa do B otrok, kot elemente vozlišča pa kopiramo po vrsti od prvega do zadnjega otroka, logični kazalec najmanjšega (prvega) niza in kazalec največjega (zadnjega) niza. Postopek ponavljamo vse do korena.



Slika 4.1: Primer zgradbe notranjega vozlišča B-drevesa nizov [8]'

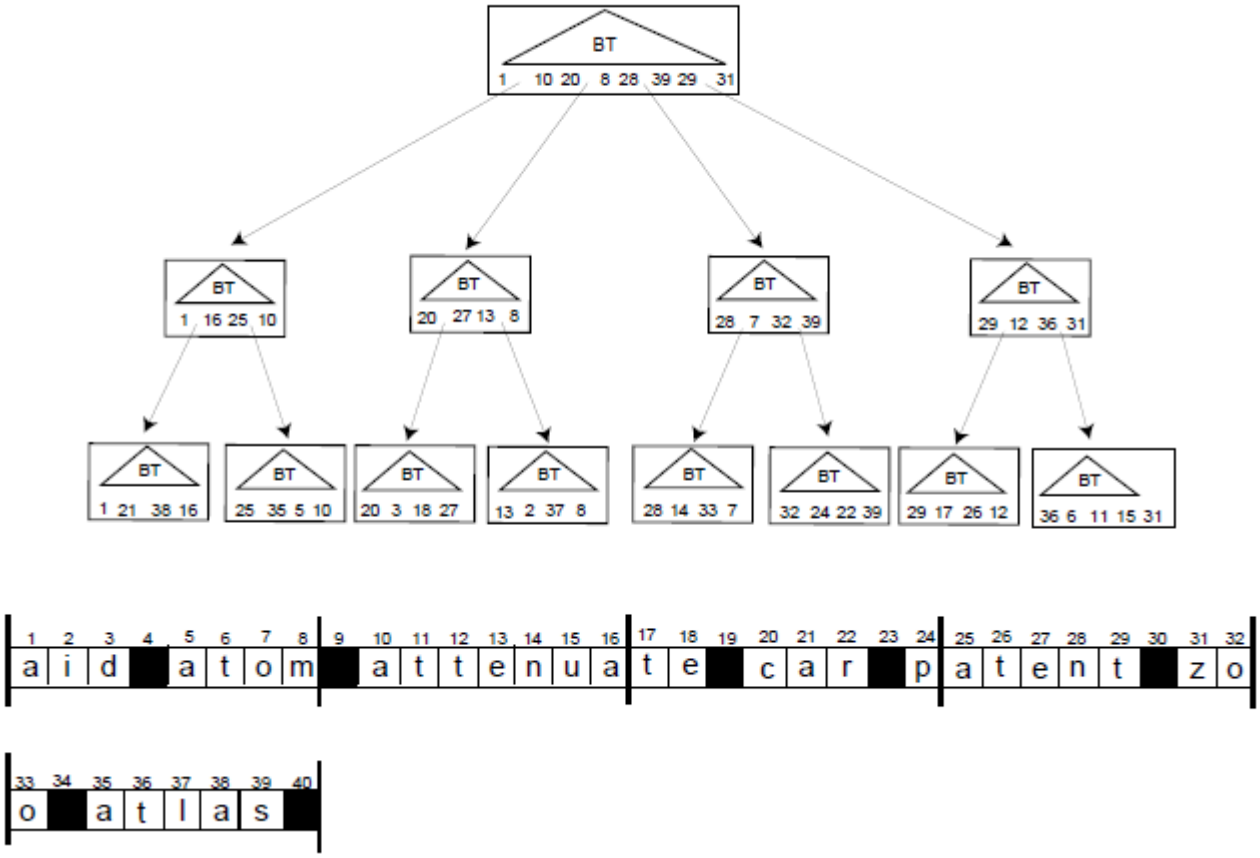
Na sliki 4.1 vidimo primer zgradbe notranjega vozlišča π. Vozlišče π ima, kot vidimo, naslednike σ_1, σ_2, ..., σ_g, elementi pa so logični kazalci najmanjših in največjih elementov v vseh naslednikih, L(σ_1), R(σ_1), L(σ_2), R(σ_2), ..., L(σ_g), R(σ_g). Tako si elementi sledijo po urejenem leksikografskem redu, prvi je najmanjši (leksikografski) element prvega naslednika (L(σ_1)), drugi največji element prvega naslednika (R(σ_1)), tretji najmanjši element drugega naslednika (L(σ_1)) in tako naprej.

Iz postopka je hitro razvidna odvisnost med številom naslednikov in številom elementov vozlišča. Ker iz vsakega naslednika kopiramo dva elementa, je elementov dvakrat več. Pravzaprav bi B-drevo nizov lahko napravili tudi tako, da bi iz vsakega naslednika kopirali samo en element (ali največjega ali najmanjšega), vendar pa bi bil algoritem iskanja bolj zakompliciran in težji.
Sedaj, ko poznamo število naslednikov in elementov vsakega vozlišča, lahko izračunamo višino drevesa H:

- $H = log_{\frac{B}{2}} k$

Z opisano strukturo že lahko rešimo problem 1. Pri reševanju problema 1, opisanega v uvodu, moramo najprej poiskati prvi niz, ki vsebuje iskano predpono, in nato še zadnji niz, ki vsebuje to predpono. Vsi nizi vmes prav tako vsebujejo iskano predpono. Trenutna struktura je podobna klasičnim B-drevesom [4].

Sedaj v to strukturo za boljšo učinkovitost operacij vpeljimo še PATRICIA drevesa [18, 8]. S pomočjo PATRICIA dreves bomo zmanjšali število primerjav med nizi za iskanje predpon znotraj posameznega vozlišča. S pomočjo PATRICIA dreves bomo potrebovali primerjavo zgolj z enim nizom, za razliko od prejšnje strukture, ko smo v najslabšem primeru potrebovali $log_2|\pi|$ primerjav. PATRICIA drevesa vključimo tako, da vsem vozliščem drevesa dodamo drevo PATRICIA, zgrajeno iz elementov vozlišča. Sedaj lahko predpone iščemo v drevesu PATRICIA in ne neposredno v polju elementov.



Slika 4.2: Primer B-drevesa nizov'

Na sliki 4.2 vidimo primer B-drevesa nizov. Drevo na primeru je sestavljeno iz nizov v tabeli na sliki, v drevo pa so vstavljeni vsi podnizi z njihovimi logičnimi kazalci. Tako je na primer niz 'aid', ki se začne na poziciji *1* in je leksikografsko najmanjši, na prvem mestu v drevesu. Drevo na primeru ima B parameter enak *4*, ker vidimo, da je maksimalno število naslednikov katerega izmed vozlišč enako *4*. Iz primera na sliki je lepo vidna lastnost B-drevesa nizov, da vsebuje starš najmanjši in največji element vseh naslednikov. Primer lahko vidimo pri korenu, prva dva elementa sta enaka *1* in *10*, ta pa sta enaka najmanjšemu in največjemu elementu prvega naslednika korena. Poleg tega na sliki vidimo še zadnjo spremembo, ki smo

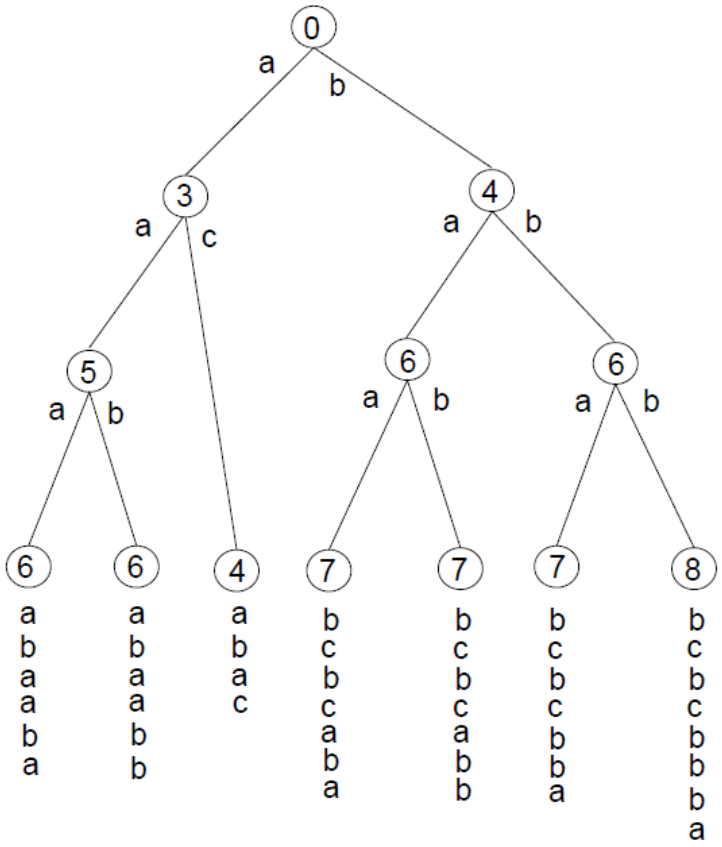
jo B-drevesom nizov dodali, in sicer PATRICIA drevesa vsakemu vozlišču. Tako vidimo, da ima vsako vozlišče drevesa PATRICIA drevo, ki je zgrajeno iz elementov vozlišča.

4.2 PATRICIA drevo

Sedaj že poznamo osnovno strukturo B-dreves nizov [8], vsakemu vozlišču pa smo dodali še PATRICIA drevo [18, 8], ki je zgrajeno iz elementov vozlišča. Za lažje razumevanje strukture PATRICIA drevesa najprej vpeljemo nekaj definicij.
Naj bo $S = \{X_1, ..., X_k\}$ množica k nizov. Dolžino skupne predpone dveh nizov X in Y označimo z L = lcp(X,Y).

- **lcp(X,Y) = L;** če X[1,L] = Y[1,L] in X[L+1] $\neq$ Y[L+1]

Gradnjo drevesa začnemo v korenu. Koren označimo z LCP, minimalno dolžino najdaljše skupne predpone med vsemi nizi. Če vemo, da so nizi leksikografsko urejeni, lahko rečemo kar $LCP = lcp(X_1, X_k)$. Korenu dodamo toliko naslednikov, kolikor je različnih znakov na poziciji $LCP + 1$. Vsako povezavo od korena pa do naslednika označimo z znakom na poziciji $LCP + 1$, ki mu pravimo zgrešeni znak (mismatched char) oziroma neujemanje. Zgrešeni znak pravimo zaradi tega, ker gre za prvi znak, ki ni skupen sosednjim nizom, torej je prvi zgrešeni znak oziroma prvo neujemanje. Postopek ponovimo v vsakem izmed naslednikov in ga ponavljamo, dokler vozlišče ne vsebuje samo enega niza. Ko vozlišče vsebuje zgolj en niz, vozlišče označimo kot list z dolžino niza, ki ga vsebuje.



Slika 4.3: Primer PATRICIA drevesa'

Za lažje razumevanje strukture si poglejmo primer na sliki 4.3. Naj bo $S = \{'abaaba', 'abaabb', 'abac', 'bcbcaba', 'bcbcabb', 'bcbcbba', 'bcbcbbba'\}$. Koren smo označili z ničlo, ker je minimalna dolžina maksimalnih skupnih predpon dolžine nič, lcp('abaaba', 'bcbcbbba') = 0. Ker imamo na prvi poziciji dva različna znaka ('a', 'b'), vstavimo dva naslednika, povezavi pa označimo z znakoma 'a' in 'b'. Na drugem nivoju imamo tako dve skupini, v prvi skupini so nizi, ki se začnejo z znakom 'a', v drugi pa tisti, ki se začnejo z 'b'. Vozlišče prve skupine označimo s *3*, ker je minimalna najdaljša dolžina skupne predpone enaka tri, lcp('abaaba', 'abac') = 3. Vozlišču nato vstavimo dva naslednika, ker imajo nizi, ki jih vozlišče vsebuje ('abaaba', 'abaabb', 'abac') dva različna znaka na poziciji *4*, povezavi pa označimo z znakoma na

tej poziciji, 'a' in 'c'. V prvem nasledniku postopek ponovimo, ker sta nam ostala še dva niza, v drugem pa je ostal samo še en niz, vozlišče spremenimo v list in ga označimo z dolžino niza, v danem primeru *4*. Postopek rekurzivno ponovimo še v drugem nasledniku korena.

4.2.1 Iskanje v PATRICIA drevesu

PATRICIA drevesa vpeljemo v strukturo B-dreves znakov zaradi tega, ker omogočajo hitro iskanje. Iskanje kličemo s funkcijo *PT-Search*. Funkcija *PT-Search* ima dva vhodna podatka, prvi parameter predstavlja iskani niz in ga označimo z P, drugi parameter pa predstavlja seznam, ki vsebuje vse nize vozlišča oziroma vse nize, ki jih vsebuje PATRICIA drevo. Rezultat funkcije je število j, ki predstavlja pozicijo niza P v trenutnem vozlišču, v leksikografski ureditvi.

Iskanje začnemo v korenu drevesa (PATRICIA drevo). Najprej si poglejmo definicijo poteka iskanja, nato pa si bomo za lažjo razumljivost pogledali primer iskanja na sliki 4.4. Iskanje vsebuje dve fazi.

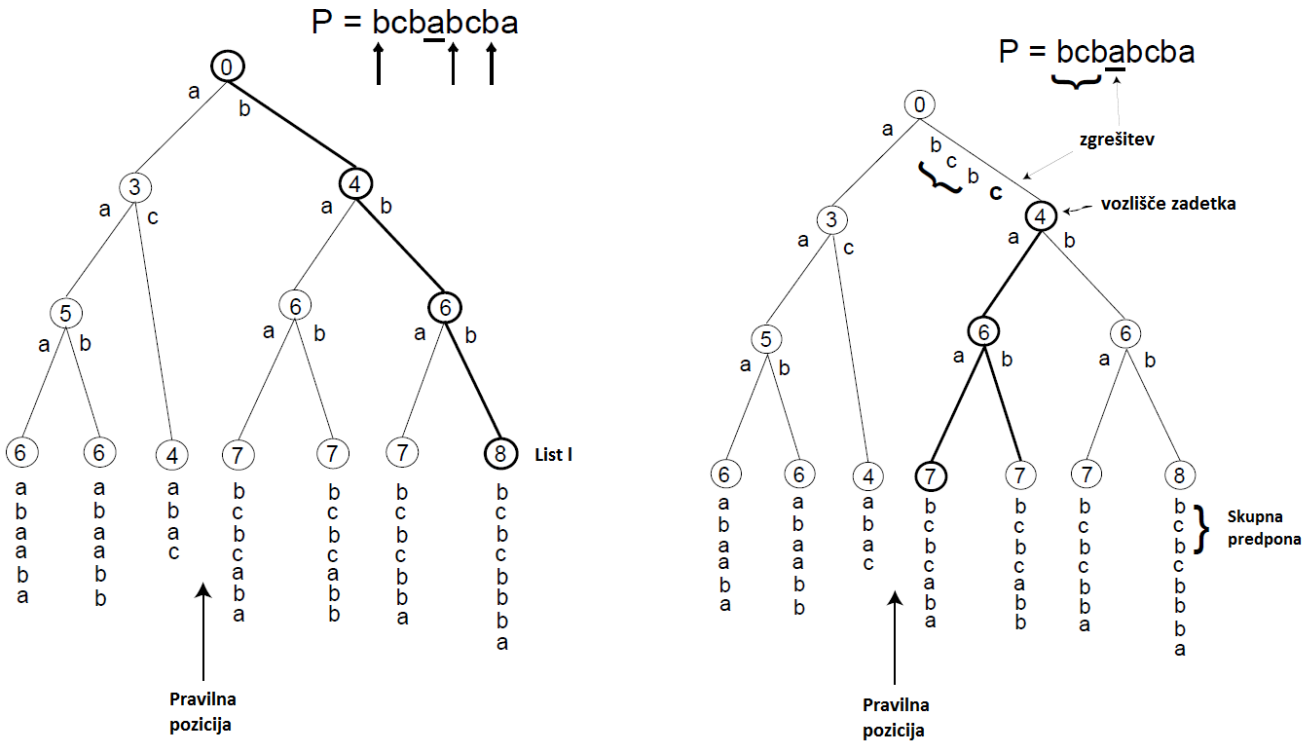
V prvi fazi potujemo od korena do lista. Ta del algoritma nam ne zagotavlja, da smo prišli v pravi list v drevesu, vendar pa pridemo v bližino pravega lista in s tem zmanjšamo območje iskanja. Trenutno vozlišče PATRICIA drevesa označimo s θ, oznako vozlišča (minimalno dolžino najdaljše skupne predpone) pa s θ_{lcp}. Vozlišče na naslednjem nivoju izberemo tako, da primerjamo zgrešene znake otrok z znakom v iskanem nizu na poziciji $\theta_{lcp} + 1$ (P$[\theta_{lcp} + 1]$). Zgrešeni znaki so leksikografsko urejeni. Premaknemo se v tisto vozlišče, v katerega znak P$[\theta_{lcp} + 1]$ sodi po leksikografski urejenosti. Kljub temu, da po koncu postopka nismo prepričani, da smo našli pravo lokacijo (pravi list θ), pa zagotovo vemo, da smo prišli v list, za katerega velja, da je $lcp(P, K_\theta) = max(lcp(P, K_i))$. Oziroma z drugimi besedami, list, v katerega smo prišli, vsebuje najdaljšo skupno predpono z iskanim nizom P.

V drugi fazi poiščemo pravo lokacijo iskanega niza. Ker smo v prvi fazi primerjali samo določene znake niza, je možno, da smo na katerem izmed

korakov napravili napako. Najprej izračunamo najdaljšo dolžino skupne predpone niza P in niza K_θ, ki jo označimo z L. L := $lcp(P, K_\theta)$.

Sedaj se po drevesu premikamo navzgor, dokler je θ_{lcp} večji od L. Najdenemu vozlišču θ pravimo vozlišče zadetka (hit node).

Nato začnemo z zadnjo fazo postopka, ko se zopet premikamo po drevesu navzdol. Če najdeno vozlišče θ ni list, primerjamo znak na poziciji L z znakom v iskanem nizu na poziciji L, P[L]. V kolikor je znak P[L] leksikografsko manjši od $K_\theta[L]$, vrnemo indeks prvega (levega) lista trenutnega vozlišča θ, sicer vrnemo indeks zadnjega (desnega) lista trenutnega vozlišča θ.



Slika 4.4: Primer iskanja v PATRICIA drevesu [8]

Na sliki vidimo prvo in drugo stopnjo algoritma iskanja v PATRICIA drevesu. Iskani niz P je enak nizu ,,bcbabcba". V levem delu slike vidimo prvo stopnjo algoritma ,,potovanje navzdol". Iskanje začnemo v korenu, ker

je L korena enak *0*, primerjamo prvi znak niza P in zgrešene znake korena. Primerjamo torej znak ,,b" in ker je drugi zgrešeni znak korena enak ,,b", se premaknemo po tej povezavi v drugi naslednik korena. Sedaj ima vozlišče oznako *4*, zato primerjamo peti (prvi znak je na poziciji *0*) znak niza P, torej ,,b". Zopet je drugi zgrešeni znak enak ,,b", zato nadaljujemo po tej povezavi. Sedaj ima vozlišče oznako *6*, zato primerjamo sedmi znak niza P, ki je zopet ,,b". Ker je ponovno oznaka druge povezave vozlišča enaka ,,b", nadaljujemo po tej povezavi. Sedaj smo prišli v list in s tem končali prvi del iskanja.

Pričnemo z drugim delom algoritma. Najprej izračunamo L = lcp(P, ,,bcbcbbba"), torej $L = 3$. Sedaj potujemo navzgor po drevesu, dokler je oznaka vozlišča večja od *3*, tako pristanemo v vozlišču z oznako *4*, ki je ,,vozlišče zadetka (hit node)". Ker je L enak *3*, sedaj primerjamo četrti znak niza P in četrtni znak skupne predpone nizov vozlišča. Ker je četrti znak skupne predpone enak ,,c", niza P pa ,,a", vrnemo kot rezultat prvi oziroma levi list vozlišča. Prvi oziroma levi list trenutnega vozlišča dobimo tako, da se od vozlišča navzdol po drevesu sprehodimo tako, da pot nadaljujemo vedno v prvem oz. levem nasledniku vozlišča. Pot nadaljujemo tako dolgo, dokler ne pridemo do lista, list v katerega smo prišli je prvi oziroma levi list vozlišča. Prvi list vozlišča je list z indeksom *4* in nizom ,,bcbcaba". Rezultat, ki ga vrnemo, je *4*.

4.3 Iskanje nizov v B-drevesih nizov

Sedaj, ko poznamo strukturo B-dreves nizov, pa si poglejmo še iskanje nizov v teh drevesih. Kot smo opisali v prvem poglavju, bomo reševali dva problema, v prvem bomo v zbirki nizov iskali tiste, ki imajo iskano predpono. V drugem problemu pa bomo v zbirki nizov iskali vse pojavitve podnizov v celotni zbirki nizov. Oba problema bomo združili in ju reševali hkrati. Združitev bomo opravili, kot smo omenili v prvem poglavju, tako, da bomo, ko bomo reševali drug primer, v drevo vstavili vse podnize nizov. Posledično bomo

lahko reševali drugi problem z iskanjem predpon, torej kot reševanje prvega problema. Primer, če bi imeli sam eno niz, 'banana', iskali pa bi vse pojavitve vzorca 'na'. V drevo bi vstavili vse podnize niza 'banana', torej bi vstavili nize 'banana', 'anana', nana', 'ana', 'na' in 'a'. Sedaj lahko iščemo pojavitve vzorca 'na' v predponah vstavljenih nizov, tako hitro najdemo dve pojavitvi niza, in sicer v tretjem in petem podnizu, 'nana' in 'na'. Torej se lahko osredotočimo samo na iskanje predpon. Od problema, ki ga želimo reševati, pa je odvisno, katere vse nize bomo vstavili v drevo.

Iskanje začnemo v korenu drevesa. Najprej iz zunanjega pomnilnika preberemo koren in ga shranimo v glavni pomnilnik in predpomnilnik. Nato v korenu poiščemo pozicijo J, za katero velja, da so vsi nizi na poziciji manjši od J, leksikografsko manjši od iskanega vzorca P in ne vsebujejo predpone P. Vsi nizi na pozicijah od vključno J pa do konca vozlišča pa leksikografsko večji ali enaki od iskanega vzorca P. Primer naj bo S enak primeru iz prejšnjega poglavja, S = {*'abaaba', 'abaabb', 'abac', 'bcbcaba', 'bcbcabb', 'bcbcbba', 'bcbcbbba'*}, P pa naj bo iskani niz, P = *'bcbcabc'*. V tem primeru bi bila pozicija J enaka 5 (za prvi niz pravimo, da je na poziciji 0), ker je niz na poziciji 4, *'bcbcabb'*, leksikografsko manjši od niza P, niz na poziciji 5 *'bcbcbba'* pa je prvi, ki je večji ali enak iskanemu nizu P. Lokacijo J bomo sicer iskali s funkcijo PT-Search, ki deluje v PATRICIA drevesu, in smo jo spoznali v prejšnjem podpoglavju.

Problem rešujemo v dveh korakih, v prvem poiščemo lokacijo predpone P v drevesu, v drugem pa poiščemo število nizov, ki vsebujejo predpono P. Kot rešitev vrnemo vse nize, ki vsebujejo iskano predpono P.

Problem oblikujemo tako, da najprej poiščemo par (τ,J), kjer τ predstavlja list drevesa, v katerega bi sodil iskani vzorec P po leksikografski ureditvi, J pa lokacijo v vozlišču. Trenutno vozlišče, v katerem iščemo iskane nize, označimo s črko π, njegove nize oz. množico nizov, ki jih vsebuje, s $S_\pi = \{K_i, ..., K_j\}$, moč množice S_π oz. število elementov vozlišča pa s $|S_\pi|$. Z oznako $L(\pi)$ in $R(\pi)$ bomo označevali levi in desni element vozlišča π.

Sedaj si lahko pogledamo psevdo kodo našega iskanja, s pomočjo katere bomo lažje razumeli, kako poteka iskanje.

Algorithm 1 Isci(P)

```
1: if P ≤_L K_1 then
2:    return (levi list vozlišča, 1)
3: else if P >_L K_k then
4:    τ := desni list vozlišča;
5:    j := |K_τ| + 1;
6:    return (τ, j)
7: else
8:    π := koren;
9:    while true do
10:      iz zunanjega pomnilnika preberi π; (S_π := (K_1....K_{2n(π)}))
11:      j := PT-Search(P, S_π);
12:      if π je list then
13:         return (π, j)
14:      else if K_j = L(σ); (kjer je σ otrok π) then
15:         τ := levi list vozlišča σ
16:         j := 1;
17:         return (τ, j);
18:      else if K_j = R(σ); (kjer je σ otrok π) then
19:         π := σ;
20:      end if
21:   end while
22: end if
```

Iz algoritma 1 lažje razumemo potek iskanja. V prvem delu algoritma (od vključno vrstice 1 do vključno vrstice 8) preverimo, ali je iskani niz manjši ali enak kot najmanjši niz v drevesu, ter ali je večji kot največji niz v strukturi. V kolikor je kateri izmed teh dveh pogojev izpolnjen, smo iskanje zaključili, in vrnemo ali prvi ali pa zadnji list drevesa ter prvo ali zadnjo

lokacijo v listu. Prvi list vozlišča (v algoritmu levi list vozlišča) je tisti, ki ga dobimo v sprehodu od trenutnega vozlišča navzdol tako, da na vsakem koraku izberemo prvega naslednika vozlišča. Sprehod rekurzivno ponavljamo tako dolgo, dokler ne pridemo v list, in to je prvi list vozlišča. V kolikor iščemo zadnji oziroma desni list, je zadeva enaka, spremenimo samo to, da se vedno premaknemo v zadnji naslednik trenutnega vozlišča.

V kolikor ni bil noben izmed prvih dveh pogojev izpolnjen, moramo iskati pozicijo niza v notranjosti strukture. Najprej se postavimo v koren drevesa in ga preberemo iz zunanjega pomnilnika. Sedaj začnemo z iskanjem, iskanje nadaljujemo tako dolgo, da smo ali prišli v list (vozlišče, ki nima otrok) ali pa je v trenutnem vozlišču najdena pozicija niza enaka 1, kar pomeni, da je iskani niz leksikografsko manjši ali enak vsem nizom vozlišča. V prvem primeru je jasno, zakaj lahko iskanje zaključimo; ko pridemo do lista, pač ne moremo nadaljevati iskanja, in vrnemo rezultat j (vrstica 15), ki predstavlja pozicijo iskanega niza znotraj trenutnega lista.

Bolj zanimiv je drugi primer, ko še nismo prišli v list, ampak je najdena pozicija j enaka 1. Takrat lahko iskanje končamo, ker iz strukture drevesa vemo, da v kolikor je niz leksikografsko manjši ali enak od prvega niza v trenutnem vozlišču, je manjši ali enak tudi od vseh nizov v poddrevesu vozlišča. In če pridemo do take situacije, vrnemo rezultat, kjer kot vozlišče vrnemo levega (prvega) naslednika vozlišča in j z vrednostjo 1 (vrstica 19).

V kolikor ni bil noben izmed danih pogojev izpolnjen, iskanje rekurzivno nadaljujemo v naslednjem nivoju drevesa.

4.4 Zahtevnost iskanja v B-drevesih nizov

Do sedaj poznamo zahtevnosti iskanja v ostalih strukturah, predstavljenih v poglavju 3. Sedaj si bomo pogledali še, kakšna je zahtevnost iskanja v predstavljeni strukturi B-dreves nizov. Zahtevnost bomo ocenjevali s številom dostopov, ki so potrebni do zunanjega pomnilnika med iskanjem, ker so ti dostopi časovno najzahtevnejši in imajo bistveni vpliv na čase iskanja.

Izkaže se, da lahko s kombinacijo B-dreves in PATRICIA dreves iščemo v nizih z zahtevnostjo $O(\frac{p+occ}{B} + log_B N)$ dostopov do diska. Kjer parameter B predstavlja velikost posameznega bloka na disku, oziroma vrednost parametra B, B-drevesa, ki še zagotavlja lastnost, da lahko vseh B elementov shranimo v posamezni blok na disku. N predstavlja vsoto velikosti vseh nizov, v katerih iščemo, p predstavlja velikost iskanega vzorca P in occ predstavlja število pojavitev vzorca P v vhodnih podatkih.

Najprej si poglejmo, kakšna bi bila zahtevnost iskanja v klasični strukturi B-dreves, ki nima PATRICIA dreves v vsakem vozlišču. Proceduro iskanja (algoritem 1) v vozlišču kličemo tolikokrat, kolikšna je višina drevesa, ki jo označimo s H. Na vsakem nivoju moram najprej prebrati vozlišče z diska, en dostop do zunanjega pomnilnika, kljub temu pa moramo v vsakem vozlišču večkrat dostopati na zunanji pomnilnik, ker imamo vozlišča predstavljena z logičnimi kazalci, kar pomeni, da moramo za vsako pridobitev niza še enkrat dostopati do zunanjega pomnilnika. Teh dostopov je potem $O(\frac{p}{B}) + 1$, ker moramo vsak niz pregledati in primerjati z iskanim vzorcem P.V najslabšem primeru moramo v vozlišču primerjati elemente z binarnim iskanjem, kar pomeni, da potrebujemo $log_2 B$ takšnih iskanj. To pomeni, da na enem nivoju v najslabšem primeru potrebujemo O($(\frac{p}{B} + 1)\ log_2 B$) dostopov do zunanjega pomnilnika. Ker pa je takšnih klicev H, dobim skupno zahtevnost v velikosti O(H$(\frac{p}{B} + 1)\ log_2 B$), kar je zagotovo manjše ali enako O($(\frac{p}{B} + 1)\ log_2 k$), kjer k predstavlja število nizov, ki jih bomo vstavili v drevo, v našem primeru je to enako N.

Zahtevnost, ki smo jo dobili, je sicer podobna, kot zahtevajo ostale strukture, kot so na primer priponska polja in priponska drevesa, vendar pa je še vedno slabša, kot smo trdili v začetku. Za izboljšanje te zahtevnosti bomo v vozlišča pripeli PATRICIA drevesa.

PATRICIA drevesa nam omogočajo, da v vsakem vozlišču primerjamo iskani vzorec P s samo enim nizom, namesto s $log_2 B$ nizov. Glavna prednost, ki nam jo PATRICIA drevesa v vozlišču omogočijo, je v tem, da za leksikografsko iskanje v vozlišču ne potrebujemo dodatnih dostopov

do diska. Ker je dovolj, da v iskanju po PATRICIA drevesu primerjamo samo zgrešene znake v drevesu z iskanim vzorcem, ne potrebujemo dodatnih dostopov. Tako nam je uspelo zahtevnost dostopov do diska zmanjšati iz $O((\frac{p}{B} + 1)\ log_2 k)$ na $O((\frac{p}{B} + 1)\ log_B k)$. Kljub velikem izboljšanju, pa še vedno nismo dosegli pričakovane zahtevnosti. Problem ostaja v ponovnem pridobivanju nizov; da bi se rešili problema, v proceduro iskanja vpeljemo parameter l, ki pove, da maksimalno dolžino skupne predpone med iskanim vzorcem P in elementi prejšnjega vozlišča. Tako lahko na naslednjem nivoju primerjamo samo znake, ki so na poziciji od l naprej. Tako potrebujemo za iskanje v vozlišču samo $\frac{l}{B} + 1$ dostopov do diska, namesto $\frac{p}{B}$.

Tako smo zmanjšali zahtevnost iskanja vseh pojavitev vzorca P na $O(\frac{p+occ}{B} + log_B k)$, kar je vsaj tako dobro, kot smo napovedali v začetku poglavja.

Poglavje 5

Implementacija:

5.1 Uporabljena orodja

5.1.1 C++

B-drevo nizov sem implementiral v programskem jeziku C++ [23]. Gre za enega izmed najpogosteje uporabljenih programskih jezikov danes. Leta 1979 je Bjerne Stroustrup v svoji doktorski disertaciji raziskoval z jezikom imenovanim Simula, ki je bi prvi predmetno usmerjen jezik. Stroustrup je hitro opazil potencial jezika za programski razvoj, kljub temu, da je bil jezik precej počasen in nepraktičen za uporabo. Kmalu je začel razvijati prvo različico programskega jezika C z razredi (C with Classes), C z razredi je bil nadgradnja osnovnega programskega jezika C z dodanimi razredi ... C z razredi je že vseboval večino tehnik, ki jih vsebuje današnji C++, od razredov, dedovanja, funkcij ... Leta 1983 je bil C z razredi preimenovan v C++. Operator "++"je v jeziku operacija povečevanja vrednosti spremenljivke in je predstavljal spremembe, ki jih je uvedel Stroustrup. Kasneje je leta 1998 organizacija za standarde ISO (international organization for standardization) postavila prvi standard C++ jezika, takrat so ga poimenovali C++98, danes je veljavni standard C++11.

Jezik se sicer razlikuje od programskega jezika C, s katerim ga ogromno

ljudi še danes zamenja. Kljub temu pa obstaja pravilo, da znajo prevajalniki večino programov iz enega jezika prevesti v drugega.

Ena izmed značilnosti programskega jezika C++ so kazalci oziroma reference, ki sicer obstajajo tudi v drugih programskih jezikih, vendar pa so v programskem jeziku C++ najbolj razširjeni. Posebnost kazalcev (*pointers*) je v tem, da omogočajo programerju bolj nadzorovano dodeljevanje pomnilniškega prostora. Če moramo v primerih brez kazalcev med funkcijami prenašati celotne kopije argumentov, pa lahko s pomočjo kazalcev prenesemo samo kazalec na lokacijo v pomnilniku, kjer se nahaja ta argument. Slaba lastnost kazalcev je nekoliko težja berljivost programov, ki jih uporabljajo, vendar pa je zaradi tega lahko prostorska in časovna zahtevnost programov manjša.

5.1.2 Valgrind

Valgrind [22] je eden izmed najbolj priljubljenih pripomočkov za analiziranje izvajanja posameznega programa. Razvit je bil z namenom iskanja zasedenosti pomnilnika med izvajanjem programov, kasneje pa se je razvil v pripomoček z ogromnim številom uporabnih možnosti. Do nedavnega je deloval samo na operacijskem sistemu Linux, na katerem je sicer še danes najbolj uporabljan. V zadnjem času so razvijalci Valgrind-a dodali še možnost izvajanja programa na operacijskem sistemu Mac OS X in na Androidu.

Idejni oče tega programskega pripomočka je Julian Seward. Program je odprtokodni in je prosto dostopen pod licenco GPL (General Public License), tako lahko izvorno kodo programa preprosto pregledamo in po želji tudi prilagodimo lastnim potrebam.

Kot smo že omenili, program vsebuje večje število programskih pripomočkov. Poleg osnovnega načina delovanja, ki analizira porabo pomnilnika med izvajanjem programa, imamo na voljo še kopico pripomočkov, recimo *Cachegrind*, *Callgrind*, *Massif*, *Hellgrind*, *exp-dhat* ... V nalogi bomo uporabili pripomoček *Cachegrind*, ki služi analiziranju uporabe predpomnilnika. Ker

bomo primerjali verzije, ki vseskozi tečejo v glavnem pomnilniku, nas bo najbolj zanimalo, koliko pri tem izkoriščamo predpomnilnik. π *Cachegrind* med izvajanje programa simulira obnašanje v izbrani pomnilniški hierarhiji in šteje pomnilniške dostope. Program predpostavlja pomnilniško hierarhijo z dvonivojskim predpomnilnikom, ki ima na prvem nivoju deljen predpomnilnik (nehomogen), na ukazni in podatkovni predpomnilnik (I1 in D1), na drugem nivoju pa homogeni predpomnilnik (L2). To pomnilniško hierarhijo predpostavlja iz preprostega razloga, ker je še danes najbolj razširjena med napravami in jo najdemo v veliki večini današnjih računalnikov. Kljub temu zna v določenih primerih *Cachegrind* avtomatsko opaziti spremembe v pomnilniški hierarhiji na nekaterih napravah, ki imajo trinivojski predpomnilnik. Na teh napravah *Cachegrind* za svoje simulacije uporabi prvi in zadnji nivo predpomnilnika. Razlog je v tem, da ima na izvajanje programa največji vpliv zadnji (najnižji) nivo predpomnilnika, ker preprečuje dostope do glavnega pomnilnika. Ne glede na to, kakšno hierarhijo *Cachegrind* predpostavi, označi prvi nivo predpomnilnika z oznakama I1 in D1 ter zadnji nivo z LL (last-level).

5.2 Opis implementacije B-drvesa nizov

Implementirali smo tri verzije. Prva verzija je klasična struktura B-drevesa znakov, kot je opisana v prejšnjem poglavju. Ta verzija je namenjena velikim zbirkam nizov, tako velikim, da ne moremo hkrati obravnavati celotne zbirke v glavnem pomnilniku, ker imamo premalo prostora. V tej verziji se najprej sestavi struktura in shrani na zunanji pomnilnik v več delih, potem pa se pri iskanju v pomnilnik prenesejo samo tisti deli, ki jih potrebujemo.

Drugi dve verziji pa sta prilagojeni in namenjeni izključno meritvam hitrosti iskanja ter primerjavam s strukturama priponskih dreves [17] in slovarja nizov [17]. Obe verziji sta namenjeni strukturam podatkov, ki niso tako velike, da jih v celoti ne bi mogli shraniti v glavni pomnilnik. Obe verziji sta zgrajeni tako, da se celotna struktura drevesa shrani neposredno v glavni

pomnilnik in se vsa iskanja izvajajo v glavnem pomnilniku.

Razlika med verzijama je v sami predstavitvi nizov v strukturi. V drugi verziji so nizi v strukturi predstavljeni na isti način kot v prvi verziji, torej z logičnimi kazalci, ki kažejo na lokacijo, kjer se niz začne. V tretji verziji pa nizi niso predstavljeni z logičnimi kazalci, ampak so neposredno shranjeni znotraj strukture. Namen druge in tretje verzije je ugotoviti, kaj je v praksi bolj pomembno, zmanjšati število ukazov ali zmanjšati porabo pomnilnika. Na eni strani imamo celotne nize v pomnilniku poleg vozlišča, kar pomeni več porabljenega prostora v pomnilniku. V drugem primeru pa porabimo manj prostora v pomnilniku in nize predstavimo z logičnimi kazalci, kar nam zagotavlja možnost upravljanja z večjim delom drevesa v predpomnilniku, ki je hitrejše.

Pri sami implementaciji sicer med verzijami ni velikih razlik, največja je pravzaprav že omenjena, in sicer v prvi in drugi verziji v samo strukturo ne shranjujemo direktnih nizov, ampak samo logične kazalce nanje (pozicije, kjer so shranjeni), medtem ko v drugi verziji namesto logičnih kazalcev shranimo kar nize. Ostale razlike pa so zgolj v tem, da druga in tretja verzija ne potrebujeta branja in zapisovanja drevesa na zunanji pomnilnik, ker je vedno shranjeno v glavnem pomnilniku.
Ogledali si bomo implementacijo prve verzije, ker je kompleksnejša od drugih dveh.
Implementacijo lahko razdelimo v dva programa:

- gradnja drevesa:
  - branje vhodnih podatkov
  - generiranje osnovne strukture b-dreves
  - gradnja PATRICIA dreves
  - zapis strukture na zunanji pomnilnik
- iskanje v drevesu:
  - branje vozlišč iz zunanjega pomnilnika

– iskanje nizov v drevesu

V drugih dveh verzijah imamo samo en sklop, ker strukture ne shranjujemo na zunanji pomnilnik. V teh dveh verzijah je potrebno ob vsakem zagonu programa, najprej zgraditi drevo, nato pa lahko iščemo v njem.

5.3 Gradnja drevesa

Za iskanje nizov moramo najprej zgraditi strukturo iz vhodnih nizov. V prvi fazi gradnje samo preberemo vse nize iz vhodne množice nizov S. Ko smo vse nize prebrali, lahko začnemo z gradnjo drevesa. Kot smo že omenili, moramo za reševanje problema 2 v drevo vnesti vse podnize vseh nizov. Najprej vstavimo vse pripone prvega niza, potem pa še vseh ostalih nizov. V algoritmu *Create String B-Tree* vidimo, kako poteka gradnja drevesa. Za vsak niz dolžine n vstavimo n elementov. Skupno pa vstavimo toliko nizov, kot je njihova skupna dolžina. Vhodni parameter algoritma, je množice nizov S, rezultat pa je zgrajeno drevo tipa ,,B-drevo znakov'.'

Algorithm 2 Create String B-Tree

```
Tree := new StringBTree();
for all s ∈ S do
  n := 0;
  while u < length(s) do
    Tree.Insert(s);
    s := s.substring(u, length(s));
  end while
end for
```

Pred začetkom gradnje drevesa izberemo vhodni parameter B, ki pove koliko elementov lahko shranimo v posamezno vozlišče. Vrednost B je zelo pomembna in mora biti nastavljena tako, da lahko vsako vozlišče shranimo na eno stran v zunanjem pomnilniku. Ta lastnost je izredno pomembna pri iskanju, kjer želimo minimalno število dostopov na zunanji pomnilnik, in ta

lastnost nam omogoči, da za vsako vozlišče potrebujemo samo eno branje na zunanji pomnilnik.

Gradnjo drevesa začnemo tako, da vstavimo prvi niz v koren drevesa, nato vstavimo še vse ostale nize. Pri vstavljanju je potrebno najti lokacijo v drevesu, kamor niz, ki ga želimo vstaviti, spada.

Vstavljanje poteka na dva načina, prvi je preprost, in sicer če je v listu, kamor mora biti niz vstavljen, prostor (list vsebuje manj kot B elementov), ga samo vstavimo na predvideno pozicijo.

Zadeva se zaplete, če je v listu, kamor bi moral biti vstavljen novi niz, že natanko B elementov. Takrat je potrebno list razdeliti, temu pravimo razcepitev. List razcepimo tako, da v trenutni list (tisti, ki bo razcepljen) vstavimo dva otroka, v prvega vstavimo prvo polovico elementov, v drugega drugo, in potem še novi element v tisti list, kamor spada glede na leksikografsko urejenost. Elemente iz vozlišča, ki je bil pravkar razcepljen, izbrišemo in vstavimo nove tako, da kopiramo najmanjšega in največjega iz prvega lista ter najmanjšega in največjega iz drugega lista.

Pred razcepitvijo pa je potrebno biti pozoren, v primeru, ko je trenutni nivo poln in je potrebno odpreti nov nivo. Struktura B-drevesa nam zagotavlja, da so vsi listi na istem nivoju, na kar je potrebno biti pozoren pri razcepitvi. Ko opravimo razcepitev, moramo hkrati vse liste prestaviti na nižji nivo, to opravimo tako, da drevo razširimo navzgor.

Posebno pozornost pri vstavljanju novega elementa je potrebno posvetiti vzdrževanju staršev vozlišč. Ob vsaki spremembi je potrebno preveriti, ali je sprememba vplivala na starša vozlišča. Kot smo omenili v opisu strukture B-dreves nizov, v poglavju 4, je ena izmed pomembnih lastnosti drevesa, da vozlišče, ki ni list, vsebuje najmanjše in največje elemente vseh naslednikov. Na to lastnost je potrebno biti pazljiv pri vsakem vstavljanju novih nizov, vedno, ko vstavimo element na prvo ali zadnje mesto v vozlišču, je potrebno popraviti elemente starša. Prav tako je potrebno, če smo spremenili prvi ali zadnji element starša, popraviti elemente starša trenutnega starša. Postopek ponavljamo dokler ali je spremenjeni element na prvi ali zadnji poziciji, ali

pa dokler ne pridemo v koren drevesa (koren drevesa je edino vozlišče, ki nima starša).

Ko vstavimo vse pripone vseh nizov v drevo, smo zgradili osnovno strukturo B-dreves nizov. Sedaj moramo v vsakem vozlišču zgraditi še PATRICIA drevo iz elementov vozlišča. Gradnjo začnemo podobno kot pri osnovni strukturi, v korenu. Koren PATRICIA drevesa označimo z L, ki predstavlja dolžino najdaljše skupne predpone prvega in zadnjega niza (leksikografsko najmanjšega in največjega). Če množico nizov, ki jih vsebuje trenutno vozlišče π označimo s $S_\pi = \{s_1, s_2, ..., s_{2n(\pi)}\}$, lahko definiramo L.

- $L_\pi := Lcp(s_1, s_{2n(\pi)})$

Nadaljujemo tako, da v vozlišče vstavimo toliko naslednikov, kolikor različnih znakov na poziciji $L + 1$ vsebujejo elementi množice S_π in vsako povezavo do naslednika označimo s tem znakom, ki mu pravimo ,,zgrešeni znak". Postopek rekurzivno ponavljamo tako, da vsako vozlišče zgradimo na tistih elementih, ki padejo v trenutno vozlišče. Postopek ponavljamo, dokler ne pridemo do enega samega elementa, takrat vozlišče označimo kot list in zaključimo gradnjo. Primer zgrajene strukture si lahko pogledamo še enkrat na sliki 4.3.

Zgrajena struktura je sedaj končana in pripravljena na iskanje. V drugi verziji gradnjo drevesa na tej točki končamo in pričnemo iskanje. V prvi verziji pa na tej točki zgrajeno strukturo shranimo in zapišemo na zunanji pomnilnik. Shranjevanje poteka po principu ,,v globino"(depth-first), tako najprej shranimo koren, nato poddrevo prvega otroka, pa poddrevo drugega, do zadnjega otroka. Vsako vozlišče shranimo na zunanji pomnilnik v binarno datoteko, z imenom najmanjšega in največjega elementa vozlišča, vmes dodamo še znak ,,-". Zaradi strukture drevesa, nimata nobeni vozlišči enakega imena, ker noben par vozlišč v drevesu ne vsebuje enakih elementov. Pri poimenovanju vozlišč napravimo posebnost samo pri korenu, ki ga poimenujemo ,,root", preprosto iz razloga, da vedno vemo, kje začeti iskanje (katero vozlišče najprej naložiti v pomnilnik). Pri shranjevanju se osredotočimo samo na pomembne lastnosti in shranimo samo tiste, ki

jih potrebujemo pri iskanju. Shranjevanje številskih in znakovnih tipov je enostavno, na disk preprosto prepišemo podatek, malo drugače pa je pri kazalcih. Kazalcev seveda ne moremo neposredno shraniti v zunanji pomnilnik. Ker gre za lokacije v glavnem pomnilniku, tam ne bi imele nobenega smisla, ko pa bi jih naslednjič prebrali, bi bili na isti lokaciji v glavnem pomnilniku verjetno že popolnoma drugi podatki.

Struktura je generirana in shranjena na zunanji pomnilnik. Sedaj moramo pred začetkom iskanja, strukturo prebrati iz zunanjega pomnilnika, nato pa lahko pričnemo z iskanjem. Seveda to velja samo takrat, ko želimo iskati v enaki zbirki nizov, takoj ko pride do kakšne spremembe v vhodni zbirki, je potrebno strukturo ali popraviti oziroma posodobiti, ali znova zgraditi. Gradnja drevesa je sicer časovno in prostorsko najzahtevnejši proces.

5.4 Iskanje v drevesu

Iskanje v drevesu sprožimo z naslednjim klicem:

- SearchSubString(P)

Vhodni parameter funkcije P predstavlja iskani vzorec, katerega pojavitve iščemo. Rezultat funkcije pa je vektor logičnih kazalcev, ki kaže na pojavitve vzorca P v vhodnih podatkih. V kolikor je rezultat prazen vektor, se vzorec P v vhodnih podatkih ne pojavi.

Prvi korak iskanja, je branje korena iz zunanjega pomnilnika. Kot smo že povedali, je koren shranjen v datoteki root, iz nje pa preberemo elemente, starša, sosede (ki jih koren sicer nima) in PATRICIA drevo. Nato lahko pričnemo z iskanjem, cilj iskanja je najti lokacijo J, kamor bi niz po leksikografski urejenosti sodil med elementi korena. Potek iskanja lokacije J smo si ogledali v algoritmu 1. Iskanje nadaljujemo v otroku z indeksom $J/2$ (ker je vsak otrok predstavljen z dvema elementoma). Sedaj ponovimo postopek, najprej iz zunanjega pomnilnika preberemo vozlišče in nato znotraj vozlišča poiščemo lokacijo J. Postopek ponavljamo ali dokler vozlišče ni list

ali pa dokler iskana pozicija J ni enaka *0*. Ko je iskana pozicija enaka nič, zaradi strukture drevesa vemo, da je element na poziciji *0* leksikografsko najmanjši element, ki ni manjši od iskanega niza. Ko smo našli lokacijo J iskanega niza, najprej preverimo, če niz na poziciji J vsebuje predpono iskanega niza; v kolikor jo, poiščemo zadnji niz, ki še vsebuje to predpono, sicer vrnemo rezultat ,,skanega niza ni v zbirki podatkov". Če niz vsebuje iskano predpono, lahko vse pojavitve predpone poiščemo preprosto tako, da primerjamo parametre *lcp* sosednjih elementov; dokler je *lcp* večji ali enak, kot je dolžina iskanega niza, je predpona prisotna. Na koncu vrnemo polje vseh nizov, ki vsebujejo iskano predpono. Najzahtevnejši element iskanja je iskanje pozicije J znotraj vozlišč, to napravimo s pomočjo PATRICIA dreves, kot smo opisali v prejšnjem poglavju.

5.5 Struktura implementacije

Sedaj bomo predstavili še tehnično plast implementacije. Program je sestavljen iz štirih razredov, dva pripadata B-drevesu, dva pa PATRICIA drevesu. Za vsako drevo imamo razred ,,drevo"in razred ,,vozlišče".

5.5.1 B-Drevo (BTree)

```
class BTree
{
  public:
    Node* nRoot;
    void WriteToFile();
    vector<int> SearchSubString(string p_SubString);

    void setRoot (Node* oldRoot, Node* newRoot)
    {
      if (oldRoot != nRoot)
      {
          throw "wrong old_root in Root::setRoot";
      }
      else
      {
          nRoot = newRoot;
      }
    }

    Node* get_root () { return nRoot; }
};
```

Slika 5.1: Struktura razreda BTree

Je osnovni razred in je starš vseh razredov. Njegova edina lastnost je vozlišče koren (root), ki je razreda Vozlišče (Node_BTree). Poleg korena ima še dve funkciji. Prva funkcija, „*WriteToFile*" je tipa void (ničesar ne vrača) in je namenjena zapisu celotnega drevesa na zunanji pomnilnik, ta funkcija se uporablja samo v prvi verziji B-dreves znakov. Druga funkcija „*SearchSubString*" je tipa vector<int> in vrne vse logične kazalce nizov, ki vsebujejo iskani vzorec P. Ta funkcija reši problem iskanja v nizih.

5.5.2 Vozlišče B-drevesa (Node_BTree)

```
class Node
{
  public:
    Node();
    vector<Node*> childrens;
    vector<int> elements;
    Node* nParent;
    PatriciaTrie ptTrie;
    Node* nPrev;
    Node* nNext;
    int lcpPrev;
    int lcpNext;
    string cMismatchedCharL;
    string cMismatchedCharR;

    void insertToATree (int p_WordIndex, int p_ElementIndex);
    void repairParent ();
    void divideNodeElements();
    void moveElement (Node* p_NewElement, bool p_FirstLast);
    void BuildPatriciaTrie();
    int findElementInsertionIndex (int p_WordIndex, int p_ElementIndex);
    vector<int> searchSubStrings(string p_SubString);
};
```

Slika 5.2: Struktura razreda Node

Je osnovni element B-drevesa nizov. Vsebuje naslednje lastnosti:

- vektor elementov tipa int ,,*elements*", ki predstavljajo logične kazalce nizov (v verziji dve neposredno nize, torej imamo vektor elementov tipa string)
- vektor kazalcev na otroke, ,,*childrens*" (isti razred)
- kazalec na vozlišče starša, ,,*nParent*'' (isti razred)
- PATRICIA drevo, ,,*ptTrie*''
- kazalca na vozlišča levega in desnega soseda, ,,*nPrev*" in ,,*nNext*" (isti razred)

- dva parametra tipa int, ki predstavljata najdaljšo skupno predpono elementov z levim in desnim sosedom, ,,*lcpPrev*" in ,,*lcpNext*"
- dva parametra tipa char, ki predstavljata prvi zgrešeni znak z levim in desnim sosedom, ,,*cMismatchedCharL*" in ,,*cMismatchedCharR*"

Večino lastnosti je uporabljenih tako pri iskanju v drevesu, kot pri gradnji drevesa. Posebnih je zadnjih šest lastnosti. Te lastnosti so namenjene izključno iskanju v drevesu, ko najdemo prvi niz, ki vsebuje željeni vzorec, lahko potem s pomočjo teh lastnosti hitreje najdemo še vse ostale pojavitve vzorca.

Poleg omenjenih lastnosti vozlišče vsebuje še tri ključne metode. Prva, ,,*insertToATree*" je namenjena vstavljanju novih elementov, in se uporabi pri gradnji drevesa. Druga, ,,*BuildPatriciaTrie*" je namenjena gradnji PATRICIA drevesa in zgradi PATRICIA drevo nad elementi vozlišča. Tretja, ,,*searchSubStrings*" pa je namenjena iskanju vzorcev ,,*p_SubString*" v vozlišču, in vrne vse nize vozlišča, ki vsebujejo iskani vzorec. Ostale metode so zgolj v pomoč glavnim trem metodam, in nimajo posebnih vlog.

5.5.3 PATRICIA drevo (PtTree)

```
class PatriciaTrie
{
  public:
    PtNode pnRoot;
    void BuildPatriciaTrie(vector<int> p_Elements);
    std::tr1::tuple<bool, int, int, int, int> SearchInPatriciaTrie(string p_SearchString);
};
```

Slika 5.3: Struktura razreda PtTree

Zelo podoben kot razred ,,Drevo"je tudi razred ,,PATRICIA drevo". Ta prav tako vsebuje samo eno lastnost, in sicer koren tipa ,,*PtNode*". Poleg te lastnosti pa tudi ta razred vsebuje dve funkciji. Prva funkcija ,,*BuildPatriciaTrie*" je namenjena gradnji drevesa, druga funkcija

„*SearchInPatriciaTrie*" pa iskanju. Prva funkcija je enako kot v razredu „*Node_BTree*" tipa void in ne vrne ničesar, je pa nekoliko bolj zanimiva druga, ki vrne trojico vrednosti. Prva vrednost je tipa int in predstavlja indeks najdene lokacije *J*, iz algoritma 1, ki nam pove, v katerem otroku moramo nadaljevati iskanje. Druga vrednost je prav tako tipa int in nam pove dolžino trenutno najdaljše skupne predpone med iskano predpono in elementom v vozlišču. Zadnja vrednost pa nam pove maksimalen indeks elementa z iskano predpono; v kolikor je ta vrednost enaka -1, pomeni, da trenutno noben element v vozlišču ne vsebuje iskane predpone, tudi ta vrednost je tipa int.

5.5.4 Vozlišče PATRICIA drevesa (PtNode)

```
class PtNode
{
  public:
    PtNode();
    PtNode* pnParent;
    int nLcp;
    vector<PtNode> nChildrens;
    vector<string> nChildsMismatchedChar;
    int ElementPointer;
    int ChildIndex;
    int LocalIndex;

    void WriteNodeToFile(FILE *p_file);
    void LoadNodeFromFile(FILE *p_file);
    void InsertElementsToPtNode(vector<int> p_Elements, int p_Min, int p_Max);
    std::tr1::tuple<bool, int, int, int> SearchInPatriciaTrie(string p_SearchString);
    PtNode* DownwardTraversal(string p_SearchString);
    PtNode* FindHitNode(int p_Lcp);
};
```

Slika 5.4: Struktura razreda PtNode

Je osnovni element PATRICIA drevesa. Vsebuje naslednje lastnosti:

- oznako vozlišča *lcp* tipa int, „*nLcp*"
- vektor otrok tipa istega razreda, „*nChildrens*"

- vektor zgrešenih znakov na poziciji *lcp + 1* tipa string, ,,*nChildsMismatchedChar*"
- kazalec na starša tipa istega razreda, ,,*pnParent*"
- parameter tipa int, ki predstavlja indeks elementa v vozlišču, ,,*ElementIndex*"

Poleg omenjenih lastnosti vozlišče vsebuje še štiri ključne metode, prva je namenjena vstavljanju novih elementov in se uporabi pri gradnji drevesa, ,,*InsertElementsToPtNode*".

Druga je namenjena zapisu vozlišča na zunanji pomnilnik v datoteko, ,,*WriteNodeToFile*".

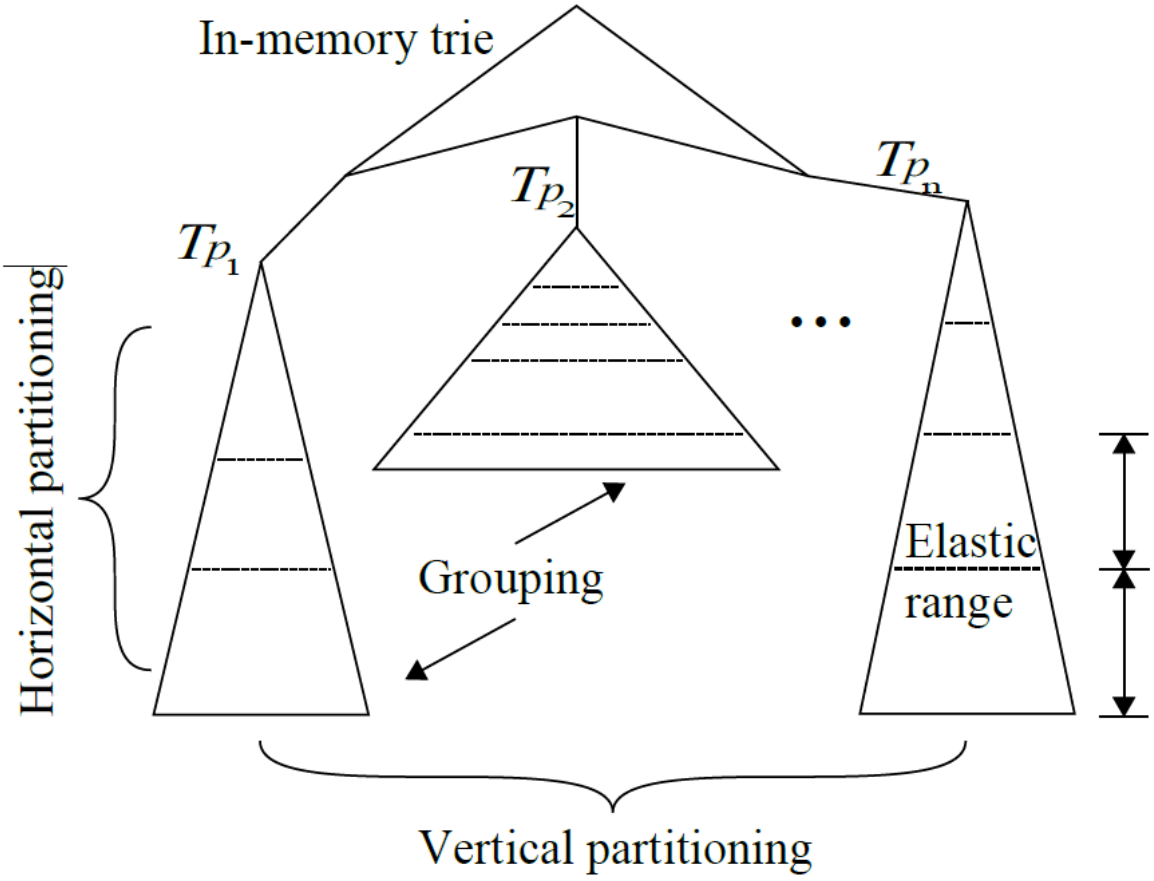
Tretja je namenjena branju vozlišča iz datoteke v glavni pomnilnik, ,,*LoadNodeFromFile*". Zadnja pa je namenjena iskanju predpon med elementi vozlišča, in vrne vse nize vozlišča, ki vsebujejo iskano predpono, ,,*SearchInPatriciaTrie*" in je ekvivalentna funkciji ,,*PT-Search*" iz algoritma 1. Ta je tudi edina bolj zanimiva, ker vrne trojko vrednosti. Prva vrednost, ki jo vrne, je tipa int in predstavlja indeks najdene lokacije *J*.

Druga vrednost je prav tako tipa int in nam pove dolžino trenutno najdaljše skupne predpone med iskano predpono in elementom v vozlišču. Zadnja vrednost je prav tako tipa int; ta vrednost pa nam pove maksimalen indeks elementa z iskano predpono. V kolikor je ta vrednost enaka -1, pomeni, da trenutno noben element v vozlišču ne vsebuje iskane predpone. To funkcijo tudi kliče funkcija ,,*searchSubStrings*" iz razreda vozlišča B-drevesa, iz slike 5.3.

5.6 ERA

ERA (*Elastic Range Algorithm*) je algoritem za gradnjo priponskih dreves, ki učinkovito deluje za zelo dolga besedila, ki jih ni mogoče shraniti v glavni pomnilnik [17]. Tako *ERA* vse podatke shrani na zunanji pomnilnik, *ERA* gradi priponsko drevo v dveh fazah, vertikalno in horizontalno. Na ta

način zmanjša rabo vhodno-izhodnih naprav, tako da dinamično prilagaja horizontalne dele neodvisno od vertikalnih. Gradnja je razdeljena na manjše podprobleme. Drevo je tako razdeljeno v več priponskih poddreves, v glavnem pomnilniku je shranjena vertikalna delitev priponskega drevesa, ki razdeli drevo na priponska poddrevesa T_p s predpono p. Za lažjo predstavitev strukture, si oglejmo naslednjo sliko 5.5, kjer je lepo vidna vertikalna in horizontalna razdelitev drevesa.



Slika 5.5: Primer strukture ERA

Ker ima uporabnik neposreden vpliv na določanje velikosti priponskih poddreves, spada algoritem v predpomnilniško zavedne algoritme, saj vnaprej

razpolaga z informacijo o velikosti razpoložljivega pomnilnika. Podrobnejšo predstavitev strukture in iskanja v strukturi najdemo v diplomski nalogi [17].

5.7 COSD

COSD (*Cache-Oblivious String Dictionary*) je algoritem za gradnjo podatkovno indeksne strukture slovar nizov [17]. Algoritem zgradi predpomnilniško nezavedni (*Cache-Oblivious*) slovar nizov in ga shrani na zunanji pomnilnik. Struktura je sicer slovar nizov, a program zgradi slovar kot korensko drevo (*cache oblivious trie*). Ko je *COSD* zgrajena, je ni mogoče spreminjati, z brisanjem ali vstavljanjem novih besed. Znotraj strukture je tako implementirana samo operacija iskanja nizov.

Algoritem je sestavljen iz štirih delov. Najprej zgradi številsko drevo T, nato ga razčleni na več manjših poddreves, ki so prek podatkovne strukture most, povezana z uravnoteženim iskalnim drevesom. Vsaka komponenta je razdeljena na vsaj eno plast. Posamezna plast komponente vsebuje žirafo in slepo drevo. Na koncu algoritem strukturo še zapiše na zunanji pomnilnik. Podrobnejšo predstavitev strukture in iskanja v strukturi najdemo v diplomski nalogi [17].

Poglavje 6

Primerjava rezultatov

Poglejmo si odgovor na ključno vprašanje, kako se B-drevo nizov izkaže v praksi?
Najprej si bomo pogledali primerjavo med implementacijama verzije dve in tri, B-drevesa znakov.

6.1 Primerjava iskanja v nizih z ostalimi strukturami

Prva struktura je seveda struktura, ki smo jo opisovali v nalogi, B-drevo nizov, drugi dve pa sta strukturi priponska drevesa, *ERA* (*Elastic Range Algorithm*) [17] in slovar nizov *COSD* (*Cache-Oblivious String Dictionary*) [17].

6.1.1 Okolje in testni podatki

Če želimo, da so rezultati primerljivi in smiselni, jih moramo pridobiti na isti napravi, pod istimi pogoji. Tehnične podrobnosti strojne opreme računalnika, na katerem bodo potekale meritve, prikazuje tabela 6.1. Računalnik sicer vsebuje *32* jeder, vendar pa tako operacija gradnje drevesa, kot iskanja v drevesu nista operaciji, ki bi ju lahko izvajali vzporedno na več

procesorjih. Tako bomo vedno izkoriščali samo eno jedro.

	Opis
Model	2 x AMD Opteron 6272 @2.10 GHz (32 jeder)
Arhitektura	x86_64
Predpomnilnik L1d	32 x 16 KB
Predpomnilnik L1i	32 x 64 KB
Predpomnilnik L2	32 x 2048 KB
Predpomnilnik L3	32 x 6144 KB
RAM	16 x 8GB DIMM DDR3 Synchronous 1333 MHz (0,8 ns)
Trdi disk	1 x 256 GB SATA
OS	Ubuntu 13.10 (GNU/Linux 3.11.0-15-generic x86_64)

Tabela 6.1: Tehnične podrobnosti strojne opreme računalnika.

Poleg identičnega okolja potrebujemo še testne podatke. Testne podatke bomo označili s črko T, sestavljeni pa so iz dveh množic, in sicer iz izvorne množice S in iskane množice X. Izvorna množica S predstavlja nize, v katerih bomo iskali, iskana množica X pa vsebuje nize, ki jih bomo iskali. Za testne podatke smo dobili podatke iz bioinformatike, v FASTA [25] formatu, ki opisujejo osnovne gradnike DNK, nukleotide. Kot izvorno množico S smo izbrali podniz nukleotida, velikost *12.000* znakov.

Kot iskano množico, pa smo ustvarili štiri različne množice nukleotidov različnih velikosti, med 1 in 12.000 znaki. Število nukleotidov znotraj posameznih množic X smo določili na 5.000, 10.000, 15.000, 20.000 nukleotidov. Iskane nize smo generirali enakomerno naključno, in sicer tako, da je bila približno polovica nizov podniz našega izvornega niza iz množice S, ostala polovica pa ne. Ker smo nize generirali naključno, dolžine med 1 in 12.000, je pričakovana dolžina posameznega niza 6.000 znakov. Cilj iskanja je poiskati vse pojavitve iskanih nukleotidov v izvorni množici S. Vsa iskanja smo obdelovali ločeno, najprej smo poiskali vse pojavitve nizov v množici X velikosti 5.000 nizov, nato množice X velikosti 10.000 nizov, nato pa še 15.000 in 20.000, vsak niz pa je bil, kot smo že omenili dolžine med 1 in

12.000 znakov.

6.1.2 Izvedba meritev

Da smo vsem strukturam zagotovili enakovredne pogoje, smo vse meritve izvedli na istem strežniku na fakulteti, opisanem v podpoglavju 6.1.1. Na naslednji sliki 6.1 lahko vidimo program napisan v programskem jeziku *C++*, s katerim smo izvedli meritve iskanj v različnih strukturah. Na sliki je sicer primer za iskanje v B-drevesu nizov, za iskanje v drugi strukturi pa se je samo spremenila vrstica *24*.

```
1  //declare variables
2  std::ifstream searchedSuffixes(sFileName.c_str());
3  string sSearchedStrings[searchedSuffixes];
4  string sTmp;
5  int nIndex = 0;
6  vector<string> vSubStringIndexes;
7
8  //read input file
9  while(std::getline(searchedSuffixes, sTmp))
10 {
11   sSearchedStrings[nIndex] = sTmp.substr(0, sTmp.size() - 1);
12   nIndex++;
13 }
14 nIndex--;
15
16 //start the timmer
17 double time_spent;
18 clock_t start, endProgram;
19 start = clock();
20
21 //search suffixes in tree
22 while(nIndex >= 0)
23 {
24   vSubStringIndexes = bTree.SearchSubString(sSearchedStrings[nIndex]);
25
26
27   nIndex--;
28 }
29
30 //stop the timmer
31 endProgram = clock();
32 time_spent = (double)(endProgram - start) / CLOCKS_PER_SEC;
33 cout << endl << stevec << endl << time_spent << endl;
```

Slika 6.1: Algoritem izvedbe meritev časovne zahtevnosti iskanja

V prvem delu programa (vrstice 1-6) definiramo spremenljivke ter branje vhodne datoteke. Za hitrejši dostop do iskanih nizov, v drugem delu, preberemo vhodno datoteko in vsak niz, ki ga bomo iskali shranimo v polje ,,*sSearchedStrings*"pod ustrezni indeks. V tretjem in zadnjem (petem) delu izvedemo časovne meritve. Najprej v tretjem delu definiramo spremenljivko tipa ,,ura"in jo poženemo, nato pa v zadnjem delu uro ustavimo ter izpišemo čas iskanja. V četrtem delu poženemo iskanje vseh nizov v drevesu. Ker je nizov veliko, iskanje ponavljamo v *while* zanki, dokler nismo preiskali vseh

nizov. Rezultate iskanja shranimo v vektor stringov, kjer so vsi nizi, ki vsebujejo iskani podniz. V kolikor je po koncu iskanja vektor prazen, vemo, da v strukturi iskani podniz ne obstaja.

Kljub temu, da smo vsem strukturam zagotovili iste podatke ter isto strojno opremo, smo vsako iskanje ponovili tisočkrat, nato pa vzeli povprečje vseh iskanj. Razlog temu je, da smo želeli zmanjšati možnost šuma na minimum, in pri večjem številu poskusov je manj možnosti, da bi povprečje imelo velika odstopanja.

Program smo prevajali s prevajalnikom ,,g++", uporabili pa smo še možnost optimizacije, ,,O3", ki nam je omogočila boljše rezultate iskanja.

Program smo izvajali na dva načina, prvi način je bil namenjen izvedbi časovnih meritev in smo ga izvedli z naslednjim klicem:

- ./StringBTree string.txt 2000 search5000.txt

Drugi način pa je bil namenjen izvedbi meritev pomnilniških dostopov in je bil izveden z naslednjim klicem:

- valgrind –tool=cachegrind –cache-sim=yes ./StringBTree string.txt 2000 search5000.txt

.

Rezultati časovnih meritev so preprosti, in sicer čas v sekundah, kako dolgo je potekalo iskanje vseh nizov v trenutni strukturi.

Bolj zakomplicirani, pa so rezultati pomnilniških dostopov, ki nam jih je vrnil *Valgrind*. Rezultati so razdeljeni v *10* kategorij, ki predstavljajo različne načine pomnilniških in predpomnilniških dostopov do podatkov in ukazov v programu. Rezultati so razdeljeni v tri skupine, tisti, ki se začnejo z znakom I, predstavljajo pomnilniške dostope do ukazov in so vedno bralnega tipa (r) (ukazov ne shranjujemo), potem so tisti, ki se začnejo z znakom D, ki predstavljajo pomnilniške dostope do podatkov. Ti dostopi so lahko tako bralni (r) kot shranjevalni (w). Zadnja vrsta pa so vrednosti LL, ki predstavljajo vsoto vseh vrednosti *I1mr*, *D1mr* in *d1mw* in tako predstavlja skupno število dostopov na zadnji nivo predpomnilnika.

6.1.3 Primerjava različnih verzij B-drevesa znakov

Preden začnemo primerjati B-drevo znakov z ostalimi strukturami, moramo ugotoviti, katera izmed verzij B-dreves znakov je v praksi najboljša. Kot smo že omenili v začetku poglavja, bomo primerjali drugo in tretjo verzijo, ki v celoti delujeta v glavnem pomnilniku in predpomnilniku. Jasno je, da bo druga verzija, ki shranjuje nize znotraj strukture, porabila več pomnilniškega prostora kot tretja, ki v strukturi vsebuje samo logične kazalce na nize. Zanima pa nas, ali bo dodatna informacija v vozliščih dovolj, da bo iskanje v drevesu hitrejše.

Najprej si poglejmo rezultate meritev pomnilniških dostopov za iskanje nizov. Rezultate dobimo s pomočjo programskega pripomočka *Valgrind*, ki smo ga predstavili v začetku poglavja. Rezultate bomo dobili na enak način, kot bomo kasneje merili čase iskanja. V obeh verzijah bomo pognali iskanje 5.000, 10.000, 15.000 in 20.000 nizov. *Valgrind* nam sicer v osnovnem izpisu vrne rezultate, ki predstavljajo vsoto vseh pomnilniških dostopov pri izvajanju programa, ker pa nas zanimajo samo tisti dostopi, ki so se zgodili v času iskanja nizov, imamo na voljo razdrobljen izpis, ki vrne število pomnilniških dostopov pri posameznih klicih funkcij. Ko dobimo število pomnilniških dostopov porazdeljenih po klicih posameznih funkcij, seštejemo števila dostopov tistih funkcij, ki se kličejo med iskanjem nizov. Rezultate dostopov pri iskanju v drugi verziji B-drevesa znakov, si lahko pogledamo v tabeli 6.2, rezultate pomnilniških dostopov pri iskanju v tretji verziji pa v tabeli 6.3.

String B-Tree v2				
	5000	**10000**	**15000**	**20000**
Ir	294.983.321	604.713.816	899.075.500	1.187.300.402
I1mr	549	13	11	13
ILmr	11	13	11	13
Dr	107.241.582	219.910.969	326.927.927	431.676.663
D1mr	652.935	1.398.643	1.999.555	2.729.449
DLmr	27.674	54.089	81.036	107.306
Dw	1.021.729	2.366.352	3.543.632	4.726.067
D1mw	15.079	20.029	29.841	59.131
DLmw	0	0	0	0
LL	668.563	1.418.685	2.029.407	2.788.593

Tabela 6.2: Rezultati meritev pomnilniških dostopov poizvedb v drugi verziji B-drevesa nizov

String B-Tree v3				
	5000	**10000**	**15000**	**20000**
Ir	295.310.698	605.368.396	900.057.672	1.188.610.095
I1mr	49.836	99.752	149.672	199.838
ILmr	11	13	11	16
Dr	107.144.055	220.055.901	327.145.371	431.966.618
D1mr	737.595	1.501.219	2.247.334	2.957.514
DLmr	24.316	47.464	70.705	93.420
Dw	1.071.729	2.436.287	3.648.584	4.866.015
D1mw	19.612	35.998	51.689	67.727
DLmw	0	0	0	0
LL	807.043	1.636.969	2.448.696	3.225.079

Tabela 6.3: Rezultati meritev pomnilniških dostopov poizvedb v tretji verziji B-drevesa nizov

Rezultati, ki nas najbolj zanimajo so *ILmr*, *DLmr* in *DLmw*, ker predstavljajo zgrešitve na zadnjem nivoju predpomnilnika. Vedno, ko pride do zgrešitve v zadnjem nivoju predpomnilnika, moramo podatke iskati v glavnem pomnilniku, kar pa je precej počasnejše kot v predpomnilniku. Iz tehničnih specifikacij [26] pomnilniškega vezja, ki je uporabljen v testnem računalniku, vidimo, da je ob zgrešitvi v zadnjem nivoju predpomnilnika potrebnih devet urinih period, da se podatki prenesejo iz glavnega pomnilnika v predpomnilnik, in da se ena urina perioda izvede v *0,8* ns (*1.333*MHz). Ob vsaki zgrešitvi na zadnjem nivoju predpomnilnika torej potrebujemo vsaj *7,2* ns.

Iz rezultatov vidimo, da druga verzija potrebuje natanko *107.317* dostopov do glavnega pomnilnika (pri iskanju *20.000* nizov), tretja verzija pa natanko *93.436* dostopov. Razlika v številu dostopov do glavnega pomnilnika med obema verzijama je torej samo *13.881* dostopov, kar skupno znese približno 100μs oz 0,0001s, kar je zelo malo za iskanje *20.000* nizov. Na drugi strani vidimo, da druga verzija potrebuje *2.788.593* dostopov na zadnji nivo predpomnilnika, tretja verzija pa *3.225.079*. Na zadnji nivo predpomnilnika mora torej tretja verzija dostopati *436.486* večkrat kot druga. Kazen dostopa na zadnji nivo predpomnilnika sicer ni tako velika, kot če je potrebno dostopati v glavni pomnilnik, vendar pa bi se lahko izkazalo, da bo zaradi veliko večjega števila teh dostopov, vseeno iskanje v drugi verziji nekoliko hitrejše kot v tretji. Časovne rezultate primerjav obeh verzij si bomo pogledali v naslednjem podpoglavju 6.1.4.

Največja razlika med obema verzijama je v številu zgrešenih ukazov na prvem nivoju predpomnilniku, posledično tudi v številu iskanih ukazov na zadnjem nivoju predpomnilnika. Razlog, da je temu tako, je v potrebi po dodatnih ukazih, ko želimo pridobiti niz, ki pripada trenutnemu vozlišču. Če želimo v drugi verziji primerjati prva dva niza trenutnega vozlišča π, s tem nimamo težav, ker imamo nize shranjene v vozliščih, jih imamo v danem trenutku na voljo in jih samo primerjamo (dodatni ukazi niso potrebni). Na drugi strani pa je v tretji verziji, ko želimo primerjati prvi in drugi niz

trenutnega vozlišča π, situacija malo drugačna. Najprej moramo pridobiti prvi niz, ker imamo na voljo samo logični kazalec, nato še drugi niz. Vsako pridobitev niza moramo sprožiti z vsaj enim ukazom, in ker je teh primerjav v samem poteku iskanja zelo veliko, je tudi teh "dodatnih"ukazov zelo veliko.

Rezultati so torej popolnoma pričakovani, v drugi verziji imamo več podatkov shranjenih v vozliščih, zato potrebujemo manj "dodatnih"ukazov, a hkrati zaradi večje porabe pomnilnika prihaja do večjega števila zgrešitev v podatkih, v tretji verziji pa ravno obratno. V nadaljevanju bomo ugotovili, katera verzija se bo v praksi izkazala kot hitrejša.

6.1.4 Izbira parametra B

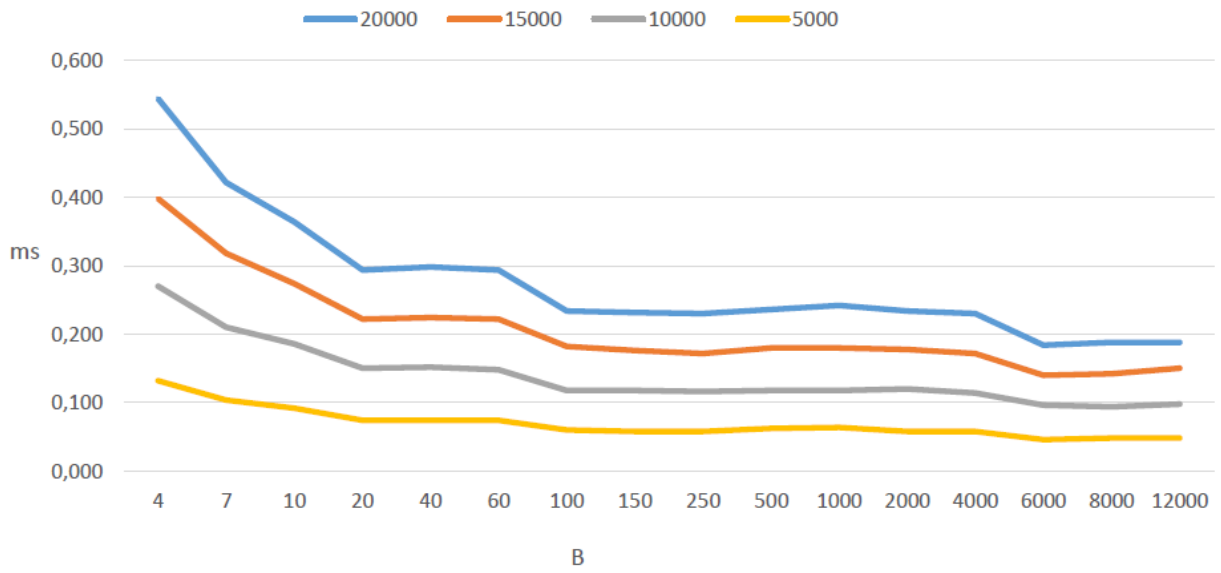
Preden začnemo poganjati testne izračune, je potrebno določiti glavni parameter, parameter B. Kot smo v prejšnjih poglavjih ugotovili, nam parameter B določi, koliko elementov shranimo v posamezno vozlišče drevesa, posredno pa nam določi tudi višino drevesa. Parameter mora biti izbran tako, da lahko vsako vozlišče shranimo na eno stran zunanjega pomnilnika (trdega diska). Na današnjih trdih diskih (HDD - hard disk drive) so strani praviloma velikosti 4 kB, torej moramo biti sposobni shraniti vsako vozlišče na velikost 4 kB.

Ker pa v teh dveh verzijah, ki ju bomo primerjali, drevesa ne shranjujemo na zunanji pomnilnik (trdi disk), lahko parameter B izberemo sami. Izbira parametra seveda ključno vpliva na iskanje, ker določa višino drevesa, razvejanost itd.

Na naslednji sliki 6.2 vidimo vpliv parametra B na čase iskanja v drugi verziji, na sliki 6.3 pa v tretji verziji. Na obeh slikah vidimo štiri različna iskanja, modra črta predstavlja iskanja v množici s *20.000* nukleotidi, oranžna črta predstavlja iskanja v množici s *15.000* nukleotidi, siva barva iskanje v množici s *10.000* nukleotidi, rumena pa iskanje v množici s *5.000* nukleotidi. Od leve proti desni, na horizontalni liniji lahko opazujemo spremembe parametra B, v naraščajočem vrstnem redu. V vertikalni smeri pa lahko opazujemo, koliko časa je program potreboval za iskanje vseh

nukleotidov. Iz vseh primerov na obeh slikah lepo vidimo, da je iskanje do določene točke vedno hitrejše, ko povečamo parameter B (in s tem zmanjšamo višino drevesa). Kasneje pa, ko pridemo do optimalne velikosti parametra, časi iskanja zopet začnejo naraščati, v kolikor nadaljujemo s povečavo parametra B. V tem primeru se izkaže, da je optimalna velikost parametra za vse štiri množice v bližini $B = 2000$, vendar pa se to lahko spremeni za različne vhodne množice podatkov.

Slika 6.2: Graf odvisnosti časov iskanja od parametra B v drugi verziji implementacije B-drevesa znakov



Slika 6.3: Graf odvisnosti časov iskanja od parametra B v tretji verziji implementacije B-drevesa znakov

Iz obeh slik lahko vidimo, da so časi iskanja optimalni nekje v okolici vrednosti, ko je B enak *2.000*, tako da bomo za meritve uporabili ta parameter. Tu je potrebno še enkrat poudariti, da nam takšna velikost ne zagotavlja, da bi lahko celotno vozlišče shranili na eno stran zunanjega pomnilnika, ki je velikosti *4* KB. A v naših verzijah to tudi ni potrebno, ker obe verziji delujeta v glavnem pomnilniku.

	verzija 2	verzija 3
5.000	0.04 s	0.058 s
10.000	0.082 s	0.12 s
15.000	0.122 s	0.178 s
20.000	0.162 s	0.234 s

Tabela 6.4: Rezultati časovnih meritev iskanj v nizih

V tabeli 6.4 vidimo, da je druga verzija v splošnem nekoliko hitrejša kot tretja, vendar so razlike zelo majhne. V tem primeru se je izkazalo, da je večje

število dostopov do zadnjega nivoja predpomnilnika bilo časovno potratnejše kot malo več dostopov do glavnega pomnilnika zaradi zgrešitve v podatkih.

6.1.5 Časovne meritve vseh poizvedb

Sedaj bomo primerjali čase iskanja vseh treh predstavljenih struktur, B-drevesa nizov, slovarja nizov COSD in priponskih dreves ERA. Meritve za algoritma ERA in COSD je pripravil kolega H. Mesić, in so povzeti iz naloge [17]. Pri meritvah nas je zanimal izključno čas, ki je potreben, da najdemo vse pojavitve vseh iskanih nizov v strukturi. Čas gradnje strukture nas v tem primeru ne zanima, kljub temu, da so načini gradnje popolnoma različni.

V tabeli 6.5 vidimo čase iskanja vseh nizov v različnih strukturah, v sekundah. Prvi stolpec tabele nam pove, koliko nukleotidov smo iskali, naslednji stolpci pa nam razkiravjo čase iskanja v posameznih strukturah.

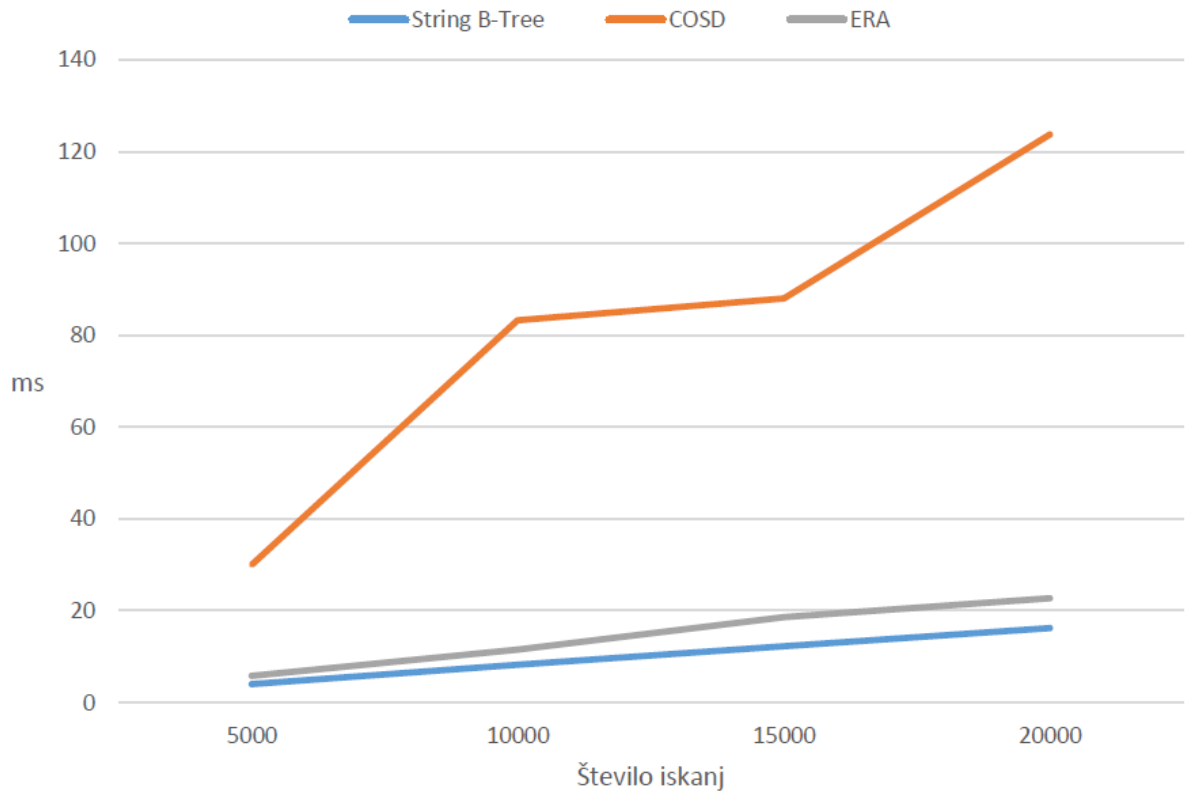
	5.000	10.000	15.000	20.000
B-drevo nizov	0.04 s	0.082 s	0.122 s	0.162 s
COSD	0.298 s	0.833 s	0.880 s	1.237 s
Era	0.058 s	0.115 s	0.186 s	0.227 s

Tabela 6.5: Rezultati časovnih meritev iskanj v nizih

Na naslednji sliki 6.4 vidimo še grafično predstavitev rezultatov. V grafu imamo štiri iskanja in sicer, iskanje v množici s po *5.000*, *10.000*, *15.000* in *20.000* nizi. V vertikalni smeri pa vidimo čase v tisočinkah sekunde, koliko časa se je posamezno iskanje izvajalo. Na sliki je lepo vidno razmerje med časi iskanj v različnih strukturah. Hitro vidimo, da sta strukturi *ERA* in B-drevesa nizov časovno zelo podobni. Presenetljivo je nekoliko hitrejša celo struktura B-drevesa nizov. Precej počasneje od obeh struktur pa so se obnesli slovarji nizov, *COSD*.

Iz slike lahko vidimo, da je iskanje v priponskih drevesih in B-drevesih nizov precej konstantno in neodvisno od vhodnih podatkov. To lastnost hitro

vidimo iz grafa, če pri B-drevesih nizov in priponskih drevesih, časi iskanja naraščajo skoraj linearno, kar pomeni, da za enkrat več iskanj potrebujemo tudi enkrat več časa, pa ta lastnost ne velja za slovarje nizov. Pri iskanju v slovarju nizov COSD, na čase iskanja vpliva tudi vhodna množica, tako lahko vidimo, da je v primeru iskanje *10.000* nizov trajalo kar *0.533* sekunde dlje kot iskanje *5.000* nizov, iskanje *15.000* nizov pa samo *0.047* sekunde dlje kot iskanje *10.000* nizov, kar je posledica vhodnih podatkov.



Slika 6.4: Časovni rezultati iskanj v strukturah

6.1.6 Meritve pomnilniških dostopov poizvedb

Obstajata dva modela algoritmov, predpomnilniško zavedni modeli, in predpomnilniško nezavedni modeli. Razlikujeta se v tem, da sta v prvem modelu velikosti I/O bloka B in predpomnilnika M znani. Algoritmi, ki

temeljijo na teh vrednostnih, so težje prenosljivi, saj se le-te razlikujejo na različnih računalniških sistemih.

Od vseh treh struktur, ki jih bomo primerjali, je tipa predpomnilniški zavedni model, samo algoritem slovarja nizov, *COSD*. V naslednjih štirih tabelah 6.6, 6.7, 6.8, 6.9 lahko vidimo primerjavo rezultatov meritev pomnilniških dostopov za iskanja 5.000, 10.000, 15.000 in 20.000 nizov. Zaradi boljše preglednosti, smo v tabele kopirali še razultate pomnilniških dostopov pri iskanju, strukture B-drevesa nizov, ki smo jih sicer predstavili že v podpoglavju 6.1.3.

	ERA	**B-drevo znakov**	**COSD**
Ir	250.337.290	294.983.321	313.462.342
I1mr	9.343	549	13
ILmr	30	11	13
Dr	63.465.975	107.241.582	60.081.292
D1mr	649.703	652.935	9.566.835
DLmr	296.007	27.674	**9.429.168**
Dw	225.669	1.021.729	722.296
D1mw	303	15.079	2.439
DLmw	0	0	0
LL	659.349	668.563	9.569.287

Tabela 6.6: Rezultati meritev pomnilniških dostopov pri iskanju 5.000 nizov

	ERA	**B-drevo znakov**	**COSD**
Ir	513.329.064	604.713.816	643.159.322
I1mr	18.653	13	13
ILmr	30	13	13
Dr	130.134.220	219.910.969	123.255.741
D1mr	1.344.792	1.398.643	19.630.966
DLmr	607.061	54.089	**19.302.277**
Dw	451.151	2.366.352	1.446.712
D1mw	564	20.029	4.896
DLmw	0	0	0
LL	1.364.009	1.418.685	19.635.875

Tabela 6.7: Rezultati meritev pomnilniških dostopov pri iskanju 10.000 nizov

	ERA	**B-drevo znakov**	**COSD**
Ir	765.802.961	899.075.500	960.435.328
I1mr	27.955	11	13
ILmr	30	11	13
Dr	194.215.306	326.927.927	184.063.639
D1mr	2.005.745	1.999.555	29.315.513
DLmr	903.608	81.036	**28.881.620**
Dw	675.974	3.543.632	2.169.920
D1mw	878	29.841	7.285
DLmw	1	0	0
LL	2.034.578	2.029.407	29.322.811

Tabela 6.8: Rezultati meritev pomnilniških dostopov pri iskanju 15.000 nizov

	ERA	**B-drevo znakov**	**COSD**
Ir	1.011.846.959	1.187.300.402	1.266.625.394
I1mr	37.225	13	13
ILmr	30	13	13
Dr	256.408.146	431.676.663	242.758.988
D1mr	2.642.741	2.729.449	38.660.812
DLmr	1.195.154	107.306	**38.008.358**
Dw	900.801	4.726.067	2.893.089
D1mw	1.119	59.131	9.701
DLmw	0	0	0
LL	2.681.085	2.788.593	38.670.526

Tabela 6.9: Rezultati meritev pomnilniških dostopov pri iskanju 20.000 nizov

Hitro vidimo, da sta strukturi priponskih dreves, *ERA* in B-drevesa nizov pričakovano precej podobni tudi pri številu pomnilniških dostopov med samimi iskanji. Struktura *ERA* je sicer bolj podobna tretji verziji implementacije B-dreves nizov, ker znotraj vozlišč nima neposredno shranjenih celotnih nizov, struktura *COSD* pa tej verziji (drugi) B-dreves znakov, ker ima znotraj vozlišč shranjene kopije nizov.

Zaradi te lastnosti, so rezultati strukture *ERA* bolj podobni tretji verziji, z večjim številom dostopov na zadnji nivo ukaznega predpomnilnika in manjšim številom dostopov do glavnega pomnilnika zaradi zgrešitev v podatkovnem predpomnilniku, a kot smo videli se te razlike nekoliko izničijo in niso tako velike.

Na drugi strani pa imamo slovar nizov *COSD*, ki ima predvsem veliko večje število zgrešitev v podatkih. Če zanemarimo razlike v številu dostopov v ukazni del predpomnilnika, vidimo, da je *COSD* potreboval kar *35.931.363* več dostopov do zadnjega nivoja predpomnilnika v primerjavi z B-drevesi nizov, in kar je še težje, kar *37.901.052* več dostopov do glavnega pomnilnika, zaradi zgrešitev v zadnjem nivoju podatkovnega predpomnilnika (pri iskanju

20.000 nizov). Posledice dodatnih ukazov, so vidne tudi pri časovnih rezultatih iskanj. Če vemo, da potrebujemo za prenos podatkov iz glavnega pomnilnika v predpomnilnik devet urinih period, in da je v našem primeru ena perioda dolga približno 0.8ns, lahko hitro izračunamo, da je algoritem COSD (pri iskanju *20.000* nizov) potreboval samo za dostope v glavni pomnilnik in premike podatkov v predpomnilnik dodatnih *0,27*s, kar je celo več, kot sta strukturi *ERA* in B-drevesa znakov potrebovali v celoti za iskanje vseh nizov. Ugotovimo lahko, da bi bil algoritem COSD popolnoma primerljiv strukturama B-dreves nizov in priponskim drevesom ERA, v kolikor ne bi imel toliko predpomnilniških zgrešitev. V tem primeru smo lepo videli, kako pomembno je upravljanje s pomnilnikom med samim izvajanjem programa.

Poglavje 7

Zaključek

Predstavljena struktura B-drevesa znakov (*String B-tree*) se je izkazala kot zelo zanimiva struktura, ki ni uporabna samo v teoriji, ampak tudi v praksi. V nalogi smo se sicer osredotočili samo na čase iskanje v strukturi, ki je zelo hitro, tako smo pri tem ignorirali še nekatere druge lastnosti, kot so čas gradnje, poraba strojne moči, pomnilnika itd. Gradnja strukture B-dreves znakov je zahteven proces, ki porabi veliko prostora in časa, prav tako je posodabljanje strukture zelo zahteven proces. V praksi je v večini primerov potrebno parameter B zmanjšati, da lahko vozlišča shranimo na eno stran zunanjega pomnilnika, kar pomeni nekoliko počasnejša iskanja v nizih.

Kljub temu je potrebno poudariti, da je struktura B-dreves znakov struktura z velikim potencialom, ki ima med drugim tudi še veliko možnosti za izboljšave; moja implementacija zagotovo ni popolna in dopušča različne optimizacije. Recimo gradnjo drevesa lahko optimiziramo, ker ob vsakem vstavljanju vemo, da bomo vstavili vse prepone posamezne besede, lahko to lastnost izkoristimo, potem bi lahko uporabili kompresijo podatkov, kar bi nam omogočilo manjšo porabo pomnilnika, in posledično večje izkoriščanje predpomnilnika. Vse te lastnosti bi lahko uporabili pri nadalnjem delu. Prav tako bi strukturo lahko preizkusili v kombinaciji z V/I napravami, recimo trdim diskom, kot zunanjim pomnilnikom.

V nalogi smo hkrati spoznali, kako pomembno je učinkovito upravljanje

s pomnilnikom in predpomnilnikom; videli, kako se je referenčni strukturi slovarja nizov maščevala lastnost, ko je preveč stvari shranjevala znotraj posameznih elementov in je zaradi tega prihajalo do večjega števila zgrešitev v predpomnilniku.

Zaključimo lahko, da je predstavljena struktura B-drevesa znakov zanimiva struktura z velikim potencialom in bi lahko bila brez težav uporabljena v različnih rešitvah problema iskanja v nizih.

Literatura

[1] A. Amir, M. Farach, Z. Galil, R. Giancarlo in K. Park "*Dynamic dictionary matching*". Journal of Computer and System Science 49, 1994 strani 111-115.

[2] A. Apostolico, "*The myriad virtues of subword trees*", Combinatorial Algorithms on Words, 1985 strani 85-96.

[3] R. Bayer, K. Unterauer "*Prefix B-trees*", ACM Transaction on Database systems 2, 1977 strani 11-26.

[4] R. Bayer, C. McCreight "*Organization and maintenance of large ordered indexes*". Acta Informatica 1, 1972 strani 173-189.

[5] D. Comer "*The ubiquitous B-Tree*". Computing Surveys 11, 1979 strani 121-137.

[6] T. H. Cormen, C. E. Leiserson, R. L. Rivest, "*Introduction to Algorithms*", MIT Press, 1990.

[7] P. Ferragina, R. Grossi, "*Fast incremental text editing*", ACAM-SIAM Symposium on Discrete Algorithms, 1995 strani 531-540.

[8] P. Ferragina, R. Grossi "*The String B-Tree: A New Data Structure for String Search in External Memory and its Applications*". Journal of the ACM 46, 1998 strani 236-280. Dne 24.4.2014 dostopno na strani `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.57.5939&rep=rep1&type=pdf`

[9] G. H. Gonnet, R. A. Baeza-Yates in T. Snider, "*Information Retrieval: Data Structures and Algorithms*", Prentice-Hall, 1992 strani 66-82.

[10] H. J. Gray, N. S. Prywes "*Outline for a multilist organized system*". Izdaja 41, sestanek ACM (Association for Computing Machinery), 1959.

[11] D. Gusfield, G. M. Landau in B. Schieber "*An efficient algorithm for all pairs suffix-prefix problem*". Information Processing Letters 42, 1992, strani 181-185.

[12] D. E. Knuth "*The Art of Computer Programming*". Addison-Wesley, 1973, poglavje 3: "*Sorting and Searching*"

[13] D. Kodek "*Arhitektura in organizacija računalniških sistemov*. Bi-Tim, 2008, strani 267-318

[14] J. Ziv, A. Lempel "*A universal algorithm for sequential data compression*". IEEE Transaction Information Theory 23, 1977 strani 337-343.

[15] U. Manber, G. Myers "*Suffix arrays: a new method for on-line string searches*". SIAM Journal on Computing 22, 1993 strani 935-948.

[16] E. M. McCreight "*A space-economical suffix tree construction algorithm*". Journal of the ACM 23, 1976 strani 262-272.

[17] H. Mesić "*Pomnilniška hirearhija in priponska drevesa pri iskanju v DNK*". Diplomsko delo, Univerza v Ljubljani, 2014.

[18] P. Morin "*Data Structures for Strings*. Advanced Data Structures, 2012, poglavje 7. Dne 26.4.2014 dostopno na strani `http://cg.scs.carleton.ca/~morin/teaching/5408/notes/strings.pdf`

[19] D. R. Morrison "*PATRICIA: Practical algorithm to retrieve information coded in alphanumeric*". Journal of the ACM 15, 1968 strani 514-534.

[20] B. Prince "*High performance memories*. J. Wiley, 1996.

[21] N. S. Prywes, H. J. Gray "*The organization of a Multilist-type associative memory*". IEEE Transaction on Communication and Electronics 68 (1963), strani 488-492.

[22] J. Seward, N. Nethercote, J. Weidendorfer in razvijalna ekipa Valgrind "*Valgrind 3.3 — Advanced Debugging and Profiling for GNU/Linux applications*. Network Theory LTD, 2008.

[23] B. Stroustrup "*The C++ Programming Language*. Addison-Wesley series in computer science, Reading 1997, third edition.

[24] P. J. Weinberger "*Unix B-trees*". Tehnični repozitorij AT&T Bell Labaratories

[25] Zhang Lab "*What us FASTA format*. University of Michigan, Zhang Lab. Dne 26.4.2014 dosegljivo na strani `http://zhanglab.ccmb.med.umich.edu/FASTA/`

[26] Samsung Electronics "*Samsung 240pin Registered DIMM based on 2Gb D-die*. Samsung DDR3L SDRAM datasheet. Dne 27.5.2014 dosegljivo na strani `http://www.samsung.com/global/business/semiconductor/file/2011/product/2011/9/6/476443ds_ddr3_2gb_d-die_based_1_35v_rdimm_rev12.pdf`